About This Information

The *Object REXX Programming Guide* describes the Object-based REstructured eXtended eXecutor, or Object REXX programming language. (When not comparing it to its traditional predecessor, we just call it REXX.) REXX is an integral part of IBM Operating System/2 (OS/2).

This information is aimed at developers familiar with OS/2 who want to use REXX to do object-oriented programming--or some mix of traditional and object-oriented programming--with the shortest learning curve possible.

This information assumes you are already familiar with the techniques of traditional structured programming, and uses them as a springboard for quickly understanding REXX and, in particular, Object REXX. This "no frills" approach is designed to help experienced programmers get involved quickly with the REXX language, exploit its virtues, and become productive fast.

What You Should Know before Reading This Information

To most effectively use this information, you should know:

- How to program with a traditional language like C, Basic, or Pascal
- How to use basic OS/2 commands for manipulating files, such as COPY, DELETE, DIR, and so on; of course, the more familiar you are with OS/2, the better

You should also have the *Object REXX Reference* on hand.

Who Should Read This Information

Anyone interested in getting a basic understanding of object-oriented concepts should read this information. Using this information, experienced programmers can learn about the REXX language and how it is like and unlike other structured programming languages. Programmers who want to broaden their programming knowledge can learn object-oriented programming with REXX. Users already experienced with REXX can learn about object-oriented programming (OO) in general, and OO programming with REXX in particular.

Programmers who want to make their applications (typically coded in C) scriptable by REXX, extend the REXX language, or control REXX scripts from other applications should consult the application programming interface (API) information in the appendixes presented here. For those who want to control REXX from *OS/2* applications, additional information on the OSA Scripting Components API is available in the *OSA Guide and Reference.*

Meet Object REXX

If you already know how to write conventional programs in a high-level language like Basic, C, or Pascal, the following information could be for you. Maybe you are familiar with REXX, maybe not. Maybe you have done some object-oriented programming and maybe you haven't. But if you want to learn more about object-oriented (OO) programming using one of the simplest, most powerful, most popular languages ever made available, you have come to the right place.

A Language with Something for Everyone

It's easy to get excited about REXX. REXX is a versatile, free-format language that is an integral part of the OS/2 operating system. Its simplicity makes it a good first language for beginners. For more experienced users and computer professionals, REXX offers powerful functions and the ability to issue commands to multiple environments.

The Main Attractions

The following aspects of REXX round out its versatility and function.

Object-Oriented Programming

Object-oriented extensions have been added to traditional REXX, but its existing functions and instructions have not changed. The Object REXX interpreter is actually an enhanced version of its predecessor, but with new support for:

- Classes, objects, and methods
- Messaging and polymorphism
- Inheritance and multiple inheritance
- CORBA-compliant access to OS/2 System Object Model (SOM) objects (CORBA is the Common Object Request Broker Architecture)

Object REXX supplies the user with a base set of classes. These are ALARM, CLASS, ARRAY, LIST, QUEUE, TABLE, SET, DIRECTORY, RELATION, BAG, MESSAGE, METHOD, MONITOR, STEM, STREAM, STRING, and SUPPLIER. Object REXX is fully compatible with earlier versions of REXX that were not object-based.

An English-like Language

To make REXX easier to learn and use, many of its instructions are meaningful English words. Unlike some programming languages that use abbreviations, REXX instructions are common words such as SAY, PULL, IF...THEN...ELSE, DO...END, and EXIT.

Fewer Rules

REXX has relatively few rules about format. A single instruction can span many lines, and you can include multiple instructions on a single line. Instructions need not begin in a particular column and you can type them in uppercase, lowercase, or mixed case. You can skip spaces in a line or entire lines. There is no line numbering.

Interpreted, Not Compiled

REXX is an interpreted language. When a REXX program runs, its language processor reads each statement from the source file and runs it, one statement at a time. Languages that are not interpreted must be compiled into object code before they can be run.

Built-In Functions and Methods

REXX has built-in functions and methods that perform various processing, searching, and comparison operations for text and numbers and provide formatting capabilities and arithmetic calculations.

Typeless Variables

REXX regards all data as objects of various kinds. Variables can hold any kind of object, so you need not declare variables as strings or as numbers.

String Handling

REXX includes capabilities for manipulating character strings. This allows programs to read and separate characters, numbers, and mixed input. REXX performs arithmetic operations on any string that represents a valid number, including those in exponential formats.

Clear Error Messages and Powerful Debugging

REXX displays messages with meaningful explanations when a REXX program encounters an error. In addition, the TRACE instruction provides a powerful debugging tool.

REXX and the OS/2 Operating System

The most vital role REXX plays is as a programming language for OS/2. A REXX program can serve as a script for the OS/2 operating system to follow. Using REXX, you can reduce long, complex, or repetitious tasks to a single command or program.

REXX is a built-in feature of OS/2, so programs are run directly from a windowed or full-screen command prompt. There is no installation process or separate environment. You can run a REXX program anywhere that you can use an OS/2 command or batch file.

A Classic Language Gets Classier

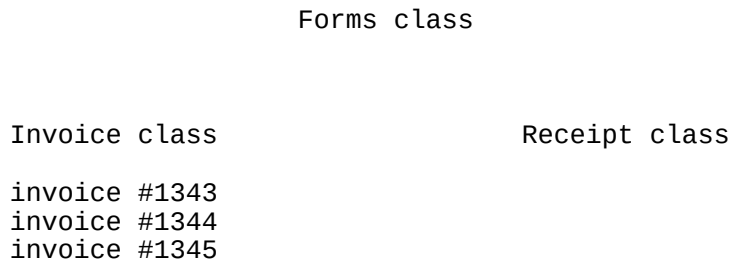
Object-oriented extensions have been added to traditional REXX without changing its existing functions and instructions. So you can continue to use REXX's procedural instructions, and incorporate objects as you become more comfortable with the technology. In general, your current REXX programs will work without change. But because Object REXX catches more errors at translate time than traditional REXX, you may have to fix these.

In object-oriented technology, *objects* are used in programs to model the real world. Similar objects are grouped into *classes*, and the classes themselves are arranged in hierarchies.

As an object-oriented programmer, you solve problems by identifying and classifying objects related to the problem. Then you determine what actions or behaviors will be required of those objects. Finally, you write the instructions to generate the classes, create the objects, and implement the actions. Your main program consists of instructions that send messages to objects.

A billing application, for example, might have an Invoice class and a Receipt class. These two classes might be members of a Forms class. Individual invoices are objects that are *instances* of the Invoice class.

Objects in a billing application



Each instance contains all the data associated with it (such as customer name, descriptions and prices of items purchased, and so on). To get at the data, you write instructions that send *messages* to the objects. These messages activate coded actions called *methods*. For an invoice object, you might want CREATE, DISPLAY, PRINT, UPDATE, and ERASE methods.

From Traditional REXX to Object REXX

How was it possible to add object-orientation to traditional REXX while maintaining compatibility? The trick was in the handling of variables. In traditional (or "classic") REXX, all data was stored as *strings.* The strings represented character data as well as numeric data. From an object-oriented perspective, we might say that traditional REXX had one kind of objects: strings. In object-oriented terminology, each string variable was an *object* that was an *instance* of the String class.

We changed REXX so that variables could reference objects other than strings. In addition to a string class, REXX now includes classes for creating arrays, queues, streams, and many others useful objects. Objects in these new REXX classes are manipulated by *methods* instead of traditional functions. To activate a method, you just send the object a *message*.

For example, instead of using the SUBSTR function on a string variable Name, you send a SUBSTR message to the string object. Here is the old way:

```
s=substr(name,2,3)
```

And here is the new way:

```
s=name~substr(2,3)
```

The tilde (~) character is the REXX *message send* operator. We call it the *twiddle*. The object receiving the message is to the left of the twiddle. The message is to the right. In this example, the Name object is sent the SUBSTR message. The numbers in parentheses (2,3) are

arguments sent as part of the message. The SUBSTR method is run for the Name object, and the result is assigned to the *s* string object.

For the new classes (array, queue, and so on), methods are provided, but not equivalent functions. For example, suppose you want to use the new REXX array object instead of the traditional string-based stem variables (such as text.1, text.2, and so on). To create an array object of five elements, you would send a NEW message to the array class as follows:

```
myarray=.array~new(5)
```

A new instance, named Myarray, of the Array class is created. (A period precedes a built-in class name in an expression, so .ARRAY is correct, not just ARRAY.) The Myarray array object has five elements. Some of the other methods, besides NEW, for array objects are PUT, AT, REMOVE, SIZE, [], and []=.

By adding object technology to its repertoire of traditional programming techniques, REXX has evolved into an object-oriented language, like Smalltalk. Object REXX accommodates the programming techniques of traditional REXX while adding new ones. With Object REXX, you can use the new technology as much or as little as you like, at whatever pace you like. You can use classic and object techniques together. You can ease into the object world gradually, building on the REXX skills and knowledge you already have.

The Object Advantage

If you are unsure about whether to employ REXX's object-oriented features, here are some tips to help you decide.

Object-oriented technology reinforces sound programming practices: hide your data from code that doesn't have to use it (encapsulation and polymorphism); partition your program in small, manageable units (classification and data abstraction); reuse code wherever possible and change it in one place (inheritance and functional decomposition), and so on.

Other advantages often associated with object technology are:

- Simplified design through modeling with objects
- Greater code reuse
- Rapid prototyping
- The higher quality of proven components
- Easier and reduced maintenance
- Cost-savings potential
- Increased adaptability and scalability

Of course, with Object REXX in particular you get REXX usability in an object-oriented language. You also get access to SOM technologies and frameworks like the OS/2 Workplace Shell, IBM Multimedia Presentation Manager, the OpenDoc architecture, and objects written in other SOM-enabled languages.

If you want to import or use OS/2 SOM objects, import or extensively manipulate the OS/2 Workplace Shell (manipulations of a subset of objects *can* be accomplished procedurally using REXX utilities), script OpenDoc parts, or create composite applications from REXX-enabled programs, Object REXX is a good way to go. It's a state-of-the-art scripting language. And OS/2 TCP/IP now provides an OS/2 Sockets API and an OS/2 FTP API for REXX, so you can script Object REXX clients and servers, and even run them on the Internet. If you've been waiting to test the waters of object-oriented programming, there couldn't be a better time or more accommodating language.

The Next Step

If you already know traditional REXX and want to go straight to the basic concepts of object-oriented programming, jump to Into the Object World.

If you are unfamiliar with traditional REXX, continue on to A Quick Tour of Traditional REXX.

A Quick Tour of Traditional REXX

Because this guide is for OS/2 programmers, we expect you are already familiar with the programming techniques of at least one other language. To speed up your involvement with REXX, this section will quickly cover the basic rules and show you how REXX is like or unlike other languages you may already know. The idea is to help you assimilate REXX more quickly and make you productive fast.

If more detail is what you want, consult the *Object REXX Reference* . It contains a full description of the REXX language. You may want to keep a copy nearby or online to look up any instruction or function not completely defined here. Think of the *Object REXX Reference* as a "technical dictionary for REXX" and this guide as "a get-you-going digest of REXX rules, key OO concepts, programming approaches, and examples."

What Is a REXX Program?

A *REXX program* is a text file, typically created using a text editor or word processor, that contains a list of instructions for your computer. REXX programs on OS/2 are interpreted, which means the program is processed line by line (like a batch file). Consequently, you don't need to compile and link REXX programs. To run a REXX program, all you need is OS/2 and the ASCII text file containing the program.

If you are familiar with programming languages such as C, Pascal, or Basic, you will find that REXX is similar. An important difference is that REXX variables have no data type and are not declared. Instead, REXX determines from context whether the variable is a string, for instance, or a number. Moreover, a variable that was treated as a number in one instruction can be treated as a string in the next.

REXX's flexibility in variable handling makes it easy for you to write programs quickly. Much of the boiler-plate instructions found in other languages is not needed. REXX keeps track of variables for you. It allocates and deallocates memory as necessary.

Another important difference is that you can execute OS/2 commands and other applications from a REXX program. This is similar to what you can do with an OS/2 Batch facility program. REXX, however, provides a more robust interface. Not only can you execute the command, you can also receive a return code from the command and use any displayed output in your REXX program. (The output normally displayed by a DIR command, for example, can be intercepted by a REXX program and used in subsequent processing.)

In addition, REXX can direct commands to environments other than the OS/2 operating system. Some applications provide an environment to which REXX can direct subcommands of the application. Some applications also provide functions that can be called from a REXX program. In these situations, REXX acts as a macro language for the application. The OS/2 Enhanced Editor is an example of an application that supports the use of REXX as a macro language.

Running a REXX Program

REXX programs on OS/2 should have a file extension of CMD, just like Batch facility files. Here is a typical REXX program named GREETING.CMD. It prompts the user to type in a name and then displays a personalized greeting:

```
/* GREETING.CMD -- a REXX program to display a greeting. */
say 'Please enter your name.'    /* Display a message    */
pull name                        /* Read response        */
say 'Hello' name                 /* Display greeting     */
exit 0                  /* Exit with a return code of 0 */
```

Notice that the program begins with a comment. In OS/2, the first line of a REXX program must start with a comment in column 1. If you omit the comment, OS/2 thinks the program is a Batch facility program, and will try to run it as such. Naturally, all sorts of errors will result.

SAY is a REXX instruction that displays a message (like PRINT in Basic or printf in C). The message to be displayed follows the SAY keyword. The single quotes are necessary to delimit a character string. In this case, the character string is *Please enter your name* . You can use double quotes (") instead of single quotes if you wish.

The PULL instruction reads a line of text from the standard input (the keyboard), and returns the text in the variable specified on the instruction. In our example, the text is returned in the variable *name* .

The next SAY instruction provides a glimpse of what can be done with REXX strings. It displays the word "Hello" followed by the name of the user, which is stored in variable *name* . REXX substitutes the value of *name* in the expression and displays the resulting string. You do not

need a separate format string as you do with C or Basic.

The final instruction, EXIT, ends the REXX program. Control returns to OS/2. EXIT can also return a value. In our example, zero is returned. The EXIT instruction is optional.

You run a REXX program just as you would run a Batch facility program. Type the file name of the program at an OS/2 command prompt. OS/2 follows its usual search rules when trying to find the program. That is, OS/2 looks for the program in the current directory. If OS/2 cannot find the command, it searches the directories listed in the PATH environment variable.

You can stop a REXX program by pressing the Control (Ctrl)+Break keys. REXX stops running the program and control returns to OS/2.

Elements of REXX

REXX programs are made up of *clauses* . Each clause is a complete REXX instruction.

REXX instructions include the obligatory program control verbs (IF, SELECT, DO, CALL, RETURN) as well as some verbs that are unique to REXX (such as PARSE, GUARD, and EXPOSE). In all, there are about 30 instructions. Many REXX programs use only a small subset of the instructions.

Complementing the instruction set is wide variety of built-in functions (over 75 of them). Many functions manipulate strings (SUBSTR, WORDS, POS, SUBWORD, and more). Other functions perform stream I/O (CHARIN, CHAROUT, LINEIN, LINEOUT, and more). Still other functions perform data conversion (X2B, X2C, D2X, C2D, and more). A quick glance through the functions section of the *Object REXX Reference* will give you an idea of the scope of capabilities available to you.

The functions that are built into the REXX language are available in REXX implementations on other operating systems. In addition to these system-independent functions, REXX on OS/2 includes a set of functions for working with OS/2 itself. These functions, known as the *REXX Utilities* , let you work with resources managed by OS/2, such as the display, the Workplace desktop, and the file system.

Instructions and functions are the building blocks of traditional REXX programs. To convert REXX into an object-oriented language, two more elements were needed: classes and methods. Classes and methods are covered in later sections. In the rest of this section we'll look at the traditional building blocks of REXX.

Writing Your Program

You can create REXX programs using any editor that can write straight ASCII files without hidden format controls. The OS/2 system editor or the Enhanced Editor are two editors that you can use.

REXX is a free-format programming language. You can indent lines and insert blank lines for readability if you wish. But even free-format languages have some rules about how language elements are used. REXX's rules center around it's basic language element: the clause.

Usually, there is one clause on each line of the program, but you can put several on a line if you wish. Just separate each clause with a semicolon (;):

```
say "Hello"; say "Goodbye"  /* Two clauses on one line */
```

To continue a clause on a second line, put a comma at the end of the line:

```
say,             /* Continuation */
"It isn't so"
```

If you need to continue a literal string, do it like this:

```
say,                /* Continuation of literal strings */
"This is a long string that we want to continue",
"on another line."
```

You'll notice that we broke the string at a convenient place (where a blank was expected). REXX automatically adds a blank. If you need to

split a string, but don't want to have a blank inserted when REXX puts the string back together, use the REXX concatenation operator (||):

```
say 'I do not want REXX to in'||,   /* Continuation with concatenation */
'sert a blank!'
```

Testing Your Program

When writing your program, you can conveniently test statements as you go along using the REXXTRY command from the OS/2 command prompt. REXXTRY is a kind of REXX mini-interpreter that checks REXX statements one at a time. If you run it with no parameter, or with a question mark as a parameter, REXXTRY will also briefly describe itself.

From your current OS/2 window, just open another window and from the command prompt type:

```
rexxtry
```

REXXTRY will describe itself and ask you for a REXX statement to test. Enter your statement; REXXTRY will run it and return any information, or display an error message if a problem is encountered. REXXTRY remembers any previous statements you have entered during the session, so to continue just type the next line in your program and REXXTRY will check it for you.

Enter an equal sign (=) to repeat your previous statement, or a question mark (?) to invoke system-provided online information about the REXX language.

When you're done, type:

```
exit
```

and press Enter to leave REXXTRY.

You can also enter a REXX statement directly on the command line for immediate processing and exit:

```
rexxtry call show
```

In this case, entering CALL SHOW displays the user variables provided by REXXTRY.

Whereas REXXTRY runs and debugs one REXX statement at a time, you can use PMREXX to do the same for an entire REXX procedure. PMREXX is a windowed application that shows, in a PMREXX window, any results displayed by your REXX procedure. You can scroll through the window using its scroll bar to view all the output.

PMREXX also provides a single line input field, so that you can supply input to the REXX procedure or to any commands called by it.

PMREXX is an OS/2 installation option; it is available if you select it during OS/2 installation. Assuming you've done this and PMREXX is available, you start it by entering PMREXX from an OS/2 command prompt, followed by the name of a REXX procedure you want to run; for example:

```
pmrexx myprog
```

You can also supply arguments for the REXX procedure. In the next example, PMREXX will run the procedure MYPROG and pass MYPROG the argument c:\test\test.data:

```
pmrexx myprog c:\test\test.data
```

PMREXX will run the REXX procedure and display any output generated by it in a scrollable output box that is part of the PMREXX window. If the procedure prompts for input, the prompt is displayed in the output box. You would then type the response in the *smaller input box* near the top of the PMREXX window.

To try it, start an OS/2 session and create this REXX procedure:

```
/* DIRTEST.CMD -- displays directory contents */
do forever
   say 'Enter the name of a directory'
```

```
   parse upper pull response
   if response='QUIT' then leave
   'dir' response
end
exit
```

Then start PMREXX again for the DIRTEST procedure:

```
pmrexx dirtest
```

When the PMREXX window is displayed, you'll see the prompt in the output box. Position the cursor in the input box, type the name of a directory, and press Enter. The contents of the directory will be listed in the output box, and another prompt will be displayed.

When the amount of displayed output exceeds the size of the output box, a slider box appears in the scroll bar.

To end the DIRTEST program, type:

```
quit
```

in the input box and press Enter.

When you run your REXX programs with PMREXX, you can use a function called RxMessageBox to display messages. This example uses it to display an error message:

```
/* ERROR.CMD -- Check the input parameter */
arg count
if count~datatype('Whole') <> 0 then do
  RxMessageBox("Argument" count "is not a whole number")
  exit
end
```

RxMessageBox displays a Presentation Manager message box titled **Error!** and an **OK** button. The REXX program waits until you click **OK**.

You can change the message box title and buttons, and add a colorful icon to the message box:

```
/* FILECHK.CMD -- Does the file exist? */
if file~Query('Exists') <> ''
  then do
    reply = RxMessageBox("Do you want to replace file",,
        "Replace File?", "YesNo", "Question")
    if reply = 7 then exit         /* user pressed 'No' */
  end
```

FILECHK.CMD displays a question in a message box with a question-mark icon and two buttons, labeled **Yes** and **No.** Clicking **Yes** or **No** gives your program the number of the button you chose. (**Yes** is 6 and **No** is 7.)

To use PMREXX as a miniature development environment for your procedures, run PMREXX in one OS/2 session and an editor in another session. Use the editor to change and save the procedure being developed. Then switch to the PMREXX session to run the modified procedure.

You can restart a procedure from PMREXX by selecting **Trace** from the action bar in the PMREXX window. If the procedure is running, select **Halt procedure** to stop it, then select **Restart** from the menu. Otherwise, just select **Restart**. The latest version of the procedure will run again.

PMREXX includes several functions useful for debugging your procedures. You can, for instance, start an interactive trace from PMREXX without having to add a TRACE instruction to your procedure. Select **Trace** on the action bar, and then select **Interactive trace on**. A check mark on the menu shows that interactive tracing is on. To stop the interactive trace, just select **Interactive trace on** again. One advantage of using the interactive trace from PMREXX is that you can turn the trace on and off while the program is running.

Once tracing is active, you can step through your procedure one clause at a time, re-do a clause that was just processed, or enter a line of REXX clauses for immediate processing. The ability to enter REXX clauses is especially useful if you want to try a fix to a problem interactively or if you want to test instruction paths that are otherwise difficult to trigger.

For example, you might want to test some error handling instructions, but cannot easily create the condition that would cause the error. By using the interactive trace, you can add REXX instructions at the right moment to fake the conditions that would cause the error handling instructions to be processed.

To process a line of one or more REXX clauses, type the line in the PMREXX input box when tracing is active and press Enter. For example, you could enter:

```
do i=1 to 10; say 'hello' i; end
```

The line is processed before the next REXX clause in the program is processed. If you simply want to step ahead to the next clause, press Enter without typing anything in the input area. You can also step ahead by selecting **Trace next clause** from the **Trace** menu. If you want to process the last REXX clause again, select **Re-do last clause** from the **Trace** menu.

To stop tracing, select **Trace off** from the **Trace** menu. This item is not selectable when the REXX procedure is waiting for user input. In this case select **Interactive trace on** again to stop the trace.

Variables, Constants, and Literal Strings

Comprehensive rules for variables, constants, and literal strings are contained in the *Object REXX Reference* . If you've programmed in Basic or C, you won't find anything too surprising about REXX's implementation.

You can name variables almost anything you want. REXX imposes few rules. A variable name can be any *symbol* (group of characters), containing up to 250 characters, with the following restrictions:

- The first character must be A-Z, a-z, !, ?, or _ . REXX translates lowercase letters to uppercase before using them.
- The rest of the characters may be A-Z, a-z, !, ?, or _, ., or 0-9.
- The period (.) has a special meaning for REXX variables. Do not use it in a variable name until you understand the rules for forming compound symbols.

Literal strings in REXX are delimited by quotation marks (either ' or "). Examples of literal strings are:

```
'Hello'
"Final result:"
```

If you need to use quotation marks within a literal string, use quotation marks of the other type to delimit the string. For example:

```
"Don't panic"
'He said, "Bother"'
```

There's another way to do this. Within a literal string, a pair of quotation marks (the same type that delimits the string) is interpreted as one of that type. For example:

```
'Don''t panic'                (same as "Don't panic"      )
"He said, ""Bother"""         (same as 'He said, "Bother"')
```

Assignments

Assignments in REXX usually take this form:

```
name = expression
```

For *name* , specify any valid variable name. For *expression* specify the information to be stored, such as a number, a string, or some calculation. Here are some examples:

```
a=1+2
b=a*1.5
c="This is a string assignment. No memory allocation needed!"
```

The PARSE instruction and its variants PULL and ARG also assign values to variables. PARSE assigns data from various sources to one or more variables according to the rules of parsing. PARSE PULL, for example, is often used to read data from the keyboard:

```
/* Using PARSE PULL to read the keyboard                          */
say 'Enter your first name and last name'   /* prompt user        */
parse pull response       /* read keyboard and put result in RESPONSE */
say response              /* possibly displays 'John Smith'       */
```

Other operands of PARSE indicate the source of the data. PARSE ARG retrieves command line arguments. PARSE VERSION retrieves the information about the version of the REXX interpreter being used. There are several other data sources that PARSE can access.

The most powerful feature of PARSE, however, is its ability to parse data according to a template that you supply. The various pieces of data are assigned to variables that are part of the template. The following example prompts the user for a date, and assigns the month, day, and year to different variables. (In a real application, you would want to add instructions to verify the input.)

```
/* PARSE example using a template */
say 'Enter a date in the form MM/DD/YY'
parse pull month '/' day '/' year
say month
say day
say year
```

The template in the above example contains two literal strings ('/'). The PARSE instruction uses these literals to determine how to split up the data.

The PULL and ARG instructions are short forms of the PARSE instruction. See the *Object REXX Reference* for lots more on REXX parsing.

Comments

In OS/2, the first line of a REXX program *must* start with a comment in column 1. This tells OS/2 your program is written in REXX, and not its built-in Batch facility. Both Batch facility and REXX programs can use the file name extension CMD. Each requires its own special processing, so OS/2 checks the first line to see which type of program it is. If it finds a REXX comment, the program is processed as REXX. Use `/*` and `*/` to mark the start and end of a comment:

```
/* This is the required column-1 comment. */

say ...  /* This is a comment on the same line as an instruction */

/* Comments may
   occupy more
   than one line. */

/***********************************
*  Comments may be boxed as headers: *
*                                   *
*  HELLO.CMD written by J. Smith     *
*        November 15, 1994           *
* A program to greet a user by name. *
***********************************/
```

For a column-1 comment it is enough to use `/* */`, but more often one uses this space to give a brief description of the program.

When REXX finds a `/*`, it stops interpreting the program until a `*/` is found, which may be a few words or several lines later.

Using Functions

REXX functions can be used in any expression. In this example, the built-in function WORD is used to return the third blank-delimited word in a string:

```
/* Example of function use                                */
myname="John Q. Public"   /* assign a literal string to MYNAME */
surname=word(myname,3)    /* assign WORD result to SURNAME     */
say surname               /* display surname                   */
```

Literal strings can be supplied as arguments to functions, so the above program can be rewritten as follows:

```
/* Example of function use                                    */
surname=word("John Q. Public",3) /* assign WORD result to SURNAME     */
say surname                      /* display surname                   */
```

Since an expression can be used on the SAY instruction, you can further reduce the program to:

```
/* Example of function use                                    */
say word("John Q. Public",3)
```

Functions can be nested. Suppose you wanted to display only the first two letters of the third word, Public. The LEFT function can return the first two letters, but you need to give it the third word. LEFT expects the input string as its first argument and the number of characters to return as its second argument:

```
/* Example of function use              */

/* Here is how to do it without nesting  */
thirdword=word("John Q. Public",3)
say left(thirdword,2)

/* And here is how to do it with nesting */
say left(word("John Q. Public",3),2)
```

Program Control

REXX has instructions such as DO, IF, and SELECT for program control. Here is a typical REXX IF instruction:

```
if a>1 & b<0 then do
    say "Whoops, A is greater than 1 while B is less than 0!"
    say "I'm ending with a return code of 99."
    exit 99
end
```

C programmers will notice that the REXX relational operator for a logical AND is different from the operator in C (which is &&). Other relational operators differ as well, so you'll want to review the appropriate section in the *Object REXX Reference* . For now, here's a list of some common comparison operators and operations:

=	True if the terms are equal (numerically, when padded, and so on)
\=, ¬=	True if the terms are not equal (inverse of =)
>	Greater than
<	Less than
<>	Greater than or less than (same as not equal)
>=	Greater than or equal to
<=	Less than or equal to
==	True if terms are strictly equal (identical)
\==, ¬==	True if the terms are NOT strictly equal (inverse of ==)

Note: Throughout the language, the NOT character, ¬, is synonymous with the backslash (\). You can use the two characters interchangeably, according to availability and personal preference. The backslash can appear in the \ (prefix not), \=, and \== operators.

A character string is taken to have the value false if it is 0, and true if it is 1. The logical operators take one or two such values (values other than 0 or 1 are not allowed) and return 0 or 1 as appropriate:

&	AND Returns `1` if both terms are true.
|	Inclusive OR Returns `1` if either term is true.
&&	Exclusive OR Returns `1` if either (but not both) is true.
Prefix \,¬	Logical NOT Negates; `1` becomes `0`, and `0` becomes `1`.

Note: On ASCII systems (for example, IBM PS/2 systems), REXX recognizes the ASCII character encoding 124 as the logical OR character. Depending on the code page or keyboard you are using for your particular country, the logical OR character may be shown as a solid vertical bar (|) or a split vertical bar (¦). The appearance of the character on your screen may not match the character engraved on the key. If you are receiving error 13, `invalid character in program`, on an instruction including a vertical bar character, make sure this character is ASCII character encoding 124.

Using the wrong relational or comparison operator is a common mistake when switching between C and REXX. You'll also notice that the familiar C language braces { } are not used in REXX for blocks of instructions. Instead, REXX uses DO/END pairs. The THEN keyword is always required.

Here is an IF instruction with an ELSE:

```
if a>1 & b<0 then do
    say "Whoops, A is greater than 1 while B is less than 0!"
    say "I'm ending with a return code of 99."
    exit 99
end
else do
    say "A and B are okay."
    say "On with the rest of the program."
end  /* if */
```

You can omit the DO/END pairs if only one clause follows the THEN or ELSE keyword:

```
if words(myvar) > 5 then
   say "Variable MYVAR has more than five words."
else
   say "Variable MYVAR has fewer than six words."
```

REXX also supports an ELSE IF construction:

```
count=words(myvar)
if count > 5 then
   say "Variable MYVAR has more than five words."
else if count >3 then
   say "Variable MYVAR has more than three, but fewer than six words."
else
   say "Variable MYVAR has fewer than four words."
```

The SELECT instruction in REXX is similar to the SELECT CASE statement in Basic and the *switch* statement in C. SELECT executes a block of statements based on the value of an expression. REXX's SELECT differs from the equivalent statements in Basic and C in that there isn't an expression following the SELECT keyword itself. Instead, expressions are placed on WHEN clauses:

```
select
when name='Bob' then
  say "It's Bob!"
when name='Mary' then
  say "Hello, Mary."
otherwise
end /* select */
```

The WHEN clauses are evaluated in order. When one of the expressions is true, the statement (or block of statements) is executed. All the other blocks are skipped, even if their WHEN clauses would have evaluated to true. (C programmers take note: statements like C's *break*

statement are not needed.)

Notice that an OTHERWISE keyword is used even though no instructions follow it. REXX doesn't require an OTHERWISE clause. However, if none of the WHEN clauses evaluates to true and you omit OTHERWISE, an error occurs. You can reduce the risk of errors by making a habit of including an OTHERWISE.

As with the IF instruction, you can use DO/END pairs for multiple clauses within SELECT cases. You don't need a DO/END pair if multiple clauses follow the OTHERWISE keyword:

```
select
when name='Bob' then
   say "It's Bob"
when name='Mary' then do
   say "Hello Mary"
   marycount=marycount+1
   end
otherwise
   say "I'm sorry.  I don't know you."
   anonymous=anonymous+1
end /* select */
```

Many Basic implementations have several different instructions for loops. In REXX, there is only the DO/END pair. All of the traditional looping variations are incorporated into the DO instruction:

```
do i=1 to 10           /* Simple loop            */
   say i
end

do i=1 to 10 by 2      /* Increment count by two */
   say i
end

b=3; a=0             /* DO WHILE -- the conditional expression */
do while a<b         /* is evaluated before the instructions   */
  say a              /* in the loop are executed.  If the      */
  a=a+1              /* expression isn't true at the outset,   */
end                  /* instructions are not executed at all.  */

a=5                  /* DO UNTIL -- like many other languages, */
b=4                  /* a REXX DO UNTIL block is executed at   */
do until a>b         /* least once.  The expression is         */
   say "Until loop"  /* evaluated at the end of the loop.      */
end
```

REXX also has a FOREVER keyword. Use the LEAVE, RETURN, or EXIT instructions to break out of the loop:

```
/* Program to emulate your 5 year old child */
num=random(1,10)  /* To emulate a three year old, move this inside the loop! */
do forever
  say "What number from 1 to 10 am I thinking?"
  pull guess
  if guess=num then do
     say "That's correct"
     leave
  end
  say "No, guess again..."
end
```

REXX also includes an ITERATE instruction that's quite handy. It skips the rest of the instructions in that iteration of the loop:

```
do i=1 to 100
   /* Iterate when the 'special case' value is reached    */
   if i=5 then iterate

   /* Instructions used for all other cases would be here */

end
```

You can use loops in IF or SELECT statements:

```
/* Say hello ten times if I is equal to 1 */
if i=1 then
```

```
   do j=1 to 10
      say "Hello!"
   end
```

Basic programmers may be wondering if there is an equivalent to the GOTO statement. There is. You can use a REXX SIGNAL instruction. SIGNAL causes control to branch to a label:

```
Signal fred;  /* Transfer control to label FRED below */
  ....
  ....
Fred: say 'Hi!'
```

As with GOTO, you need to be careful about how you're using SIGNAL. In particular, you should not try to signal into the middle of a DO/END block or into a SELECT structure.

Subroutines and Procedures

In REXX you can write routines that make all variables accessible to the called routine. You can also write routines that hide the caller's variables.

Let's start with an example of a routine in which all variables are accessible:

```
/* Routine example                    */
i=10               /* Initialize I    */
call myroutine     /* Call routine    */
say i              /* Displays 22     */
exit               /* End main program */

myroutine:         /* Label           */
   i=i+12          /* Increment I     */
return
```

The CALL instruction calls routine MYROUTINE. A label (note the colon) marks the start of the routine. A RETURN instruction ends the routine. Notice that an EXIT statement is required in this case to end the main program. If EXIT is omitted, REXX assumes that the following instructions are part of your main program and will execute those instructions, producing interesting yet undesirable results. The SAY instruction displays 22 instead of 10 because the caller's variables are accessible to the routine.

You can return a result to the caller by placing an expression on the RETURN instruction, like this:

```
/* Routine with result                */
i=10               /* Initialize I    */
call myroutine     /* Call routine    */
say result         /* Displays 22     */
exit               /* End main program */

myroutine:         /* Label           */
return i+12        /* Increment I     */
```

The returned result is available to the caller in the special variable RESULT, as shown above. If your routine returns a result, you can call it as a function:

```
/* Routine with result called as function  */
i=10               /* Initialize I         */
say myroutine()    /* Displays 22          */
exit               /* End main program     */

myroutine:         /* Label                */
return i+12        /* Increment I          */
```

You can pass arguments to this sort of routine, although there isn't much to be gained by doing so. All variables are available to the routine anyway.

Let's move on to routines that separate the caller's variables from the routine's variables. Except for the smallest programs, you'll want to use this format. It eliminates the risk of accidentally writing over a variable used by the caller or by some other unprotected routine. To get

protection, use the PROCEDURE instruction, as follows:

```
/* Routine example using PROCEDURE instruction                          */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results */
do i=1 to 100
   call cointoss                                    /* Flip the coin      */
   if result='HEADS' then headcount=headcount+1     /* Increment counters */
   else tailcount=tailcount+1
                                                    /* Report results     */
   say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                                /* End main program   */

cointoss: procedure             /* Use PROCEDURE to protect caller         */
   i=random(1,2)                /* Pick a random number: 1 or 2            */
   if i=1 then return "HEADS"   /* Return English string                   */
return "TAILS"
```

In the above example, the variable I is used in both the main program and in the routine. When the PROCEDURE instruction is placed after the routine label, the routine's variables become local variables. They are isolated from all other variables in the program. Without the PROCEDURE instruction in the above example, the program would loop indefinitely. On each iteration the value of I would be reset to some value less than 100, which means the loop would never end. (If a programming error causes your procedure to loop indefinitely, use Ctrl+Break or close the OS/2 session to end the procedure.)

To access variables outside of the routine, add an EXPOSE operand to the PROCEDURE instruction. List the desired variables after the EXPOSE keyword:

```
/* Routine example using PROCEDURE instruction with EXPOSE operand      */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results                                */
do i=1 to 100
   call cointoss                                    /* Flip the coin      */
   say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                                /* End main program   */

cointoss: procedure expose headcount tailcount /* Expose the counter variables */
   if random(1,2)=1 then do                    /* Pick a random number: 1 or 2 */
      headcount=headcount+1                    /* Bump counter...              */
      return "HEADS"                           /* ...and return English string */
   end
   else
      tailcount=tailcount+1
return "TAILS"
```

To pass arguments to a routine, separate the arguments with commas on the call:

```
call myroutine arg1, "literal arg", arg3   /* Call as subroutine */
myrc=myroutine(arg1, "literal arg", arg3)  /* Call as function   */
```

In the routine, use the PARSE ARG instruction to retrieve the argument.

Into the Object World

Object REXX includes features typical of an object-oriented language--features like subclassing, polymorphism, and data encapsulation. As we said at the outset, Object REXX is an *extension* of the traditional REXX language, which has been expanded to include classes, objects, and methods. These extensions do not replace traditional REXX functions, or preclude the development or running of traditional REXX programs. You can program as before, program with objects, or intermix objects with regular REXX instructions. *You* decide when to use REXX's object-oriented features. The REXX programming concepts that support these features are the subject of this section.

What Is Object-Oriented Programming?

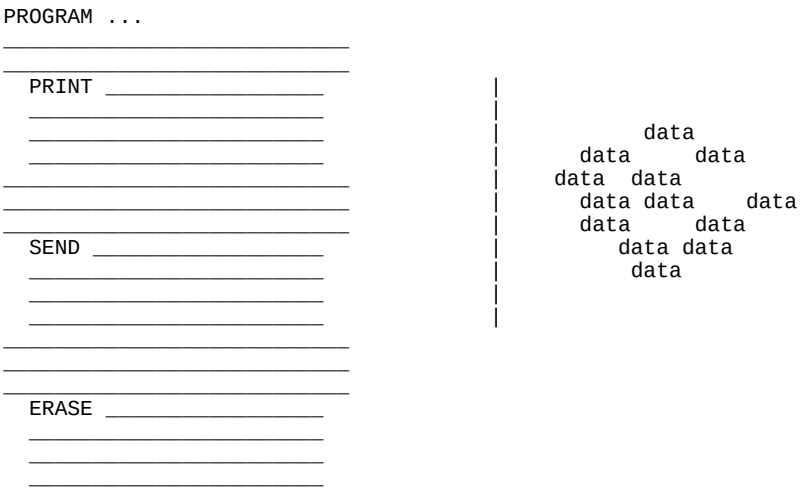
Object-oriented programming is a way to write computer programs by focusing not on the instructions and operations a program uses to manipulate data, but on the data itself. First, the program simulates, or models, objects in the physical world as closely as possible. Then the objects interact with each other to bring the desired result.

It is not unusual for computer data to represent things in the physical world--a company's employees, money in a bank account, a report. Real-world objects are stored as data so the computer can *act* upon it to some purposeful end. For example, when you "PRINT a report," *PRINT* is the action and *report* is the object acted upon. Often multiple actions apply; you could also SEND the report or ERASE the report.

Modularizing Data

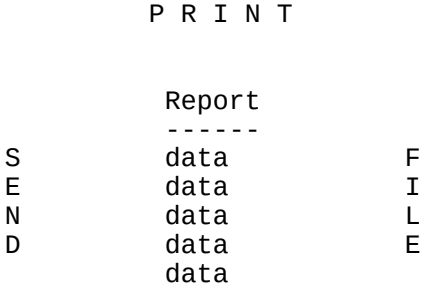
In conventional, structured programming, actions like PRINT are often isolated from the data by placing them in subroutines or modules. A module typically contains an operation for implementing one simple action. You might have a PRINT module, a SEND module, an ERASE module. These actions are independent of the data they operate on.

Modular program with isolated data



But with object-oriented programming, it is the data that is modularized. And each data module includes *its own* operations for performing actions directly related to its data.

Modular data--a report object



```
          E R A S E
```

In the case of *report,* the *report* object would contain its own built-in PRINT, SEND, ERASE, and FILE operations.

Object-oriented programming lets you model real-world objects--potentially very complex ones--precisely and elegantly. As a result, object manipulation becomes easier within the computer's programming. Computer instructions become simpler, and can be modified later with minimal effort.

Object-oriented programming *hides* any information that need not be known in order to act on an object, thereby concealing the object's complexities. Complex tasks can then be initiated simply, at a very high level. It's like moving your arm. To do so, you do not have to know about the many electrical, chemical, and mechanical interactions that must occur in the bones, muscles, tissues, and systems involved. These operations occur at lower levels. You just *move your arm.*

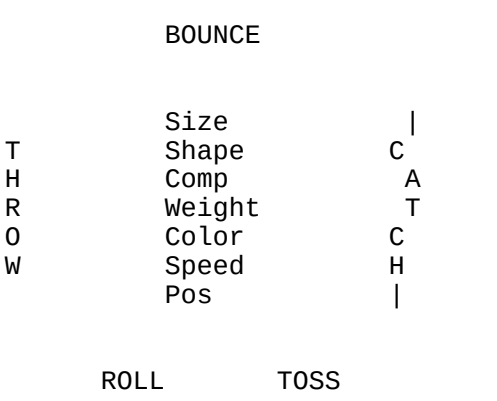
Objects Model Things in the Real World

In object-oriented programming, objects are modeled to real-world objects. A real-world object has:

- Actions related to it
- Characteristics of its own

Take a ball, for instance. A ball can be acted on--rolled, tossed, thrown, bounced, caught. But it also has its own physical characteristics--size, shape, composition, weight, color, speed, position. An accurate data model of a real ball would define not just the physical characteristics of size, shape, and the rest. Rather, this is what conventional programming would do. Instead it would define *all* related actions and characteristics in one complete "package."

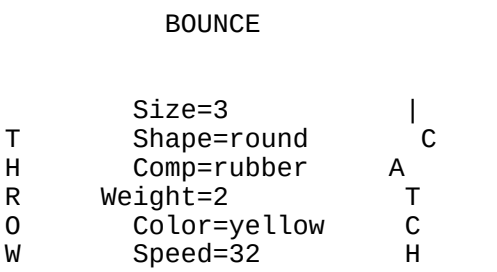
A ball object

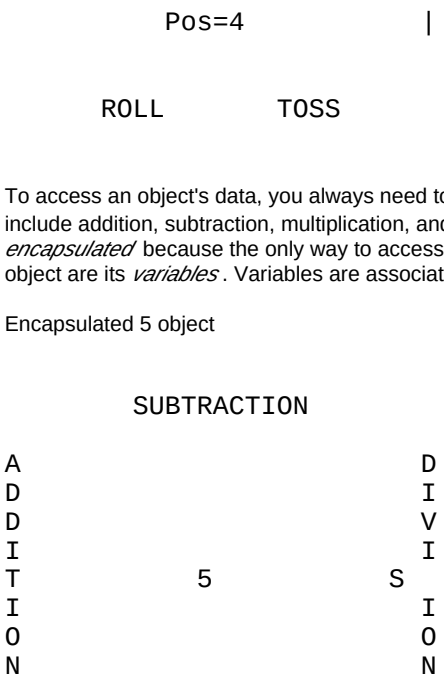


In object-oriented programming, objects are the basic building blocks--the fundamental units of data.

There are many kinds of objects; for example, character strings, collections, and input and output streams. An object--such as a character string--always consists of two parts: the possible actions or operations related to it, and its characteristics, or variables (a variable has a variable *name,* and an associated data value that can change over time). These actions and characteristics are so closely associated that they cannot be separated.

Ball object with variable names and values





To access an object's data, you always need to specify an action. For example, suppose the object is the number 5. Its actions might include addition, subtraction, multiplication, and division. Each of these actions is an interface to the object's data. The data is said to be *encapsulated* because the only way to access it is through one of these surrounding actions. The encapsulated internal characteristics of an object are its *variables*. Variables are associated with an object and exist for the lifetime of that object.

Encapsulated 5 object

REXX comes with a basic set of classes for creating objects (see The Basics of Classes). Therefore, you can devise objects that exactly match the needs of a particular application.

How Objects Interact

You have seen that the actions within an object are its only interface to other objects. Actions form a kind of "wall" that encapsulates the object, and shields its internal information from outside objects. This shielding is called *information hiding.* Information hiding protects an object's data from corruption by outside objects, and also protects outside objects from relying on another object's private data, which may change without warning. Because an object's actions are its only interface to the outside, one object can act upon another (or cause it to act) only by calling that object's actions. How do they do it? They send *messages.*

OO programming in REXX, then, consists of sending messages to objects.

Objects respond to these messages by performing an action, returning some data, or both. A message to an object must specify:

- A receiving object
- The "message send" symbol (~), sometimes called the "twiddle"
- The action and, optionally in parentheses, any parameters it needs to perform its function

So the message format looks like this:

```
object~action(parameters)
```

Say the object is the string `!iH.` Sending it a message to use its REVERSE action:

```
'!iH'~reverse
```

returns the string object `Hi!.`

Methods Are "Coded Actions"

Sending a message to an object results in performing some action; that is, it results in running some underlying code. The action-generating code is called a *method*. When you send a message to an object, you specify its method name in the message. (Method names are simply character strings, like "REVERSE.") Doing this runs the method with the corresponding name. In the preceding example, sending the `reverse` message to the `!iH` object causes it to run the REVERSE method. Most objects are capable of more than one action, and so have a number of available methods.

The classes REXX provides include their own predefined methods. The Message class, for example, has the COMPLETED, INIT, NOTIFY, RESULT, SEND, and START methods. When you create your own classes, you can write new methods for them in REXX code. Much of the object programming in REXX is writing the code for the methods you create.

Polymorphism

REXX lets you send the same message to objects that may be dissimilar:

```
'!iH'~reverse  /* Reverses the characters "!iH" to form "Hi!"  */
pen~reverse     /* Reverses the direction of a plotter pen      */
ball~reverse    /* Reverses the direction of a moving ball      */
```

As long as the dissimilar objects (like the preceding `!iH`-string, pen, and ball) each have their own REVERSE method corresponding with the message, REVERSE will run even though the programming implementation may be very different for each object. This ability to hide different functions behind a common interface is called *polymorphism*. As a result of information hiding, each object in the above example knows only its own version of REVERSE. And even though the objects are dissimilar, each will reverse itself as dictated by its own code.

Although the `!iH` object's REVERSE code is different from the plotter pen's, the method name can be the same because REXX keeps track of the methods each object owns. The ability to reuse the same method name so that one message can initiate more than one function is another feature of polymorphism. You don't need to have multiple message names like REVERSE_STRING, REVERSE_PEN, REVERSE_BALL, REVERSE_*anyobject.* This keeps method-naming schemes simpler and makes complex programs easier to follow and modify.

The ability to hide the various implementations of a method while leaving the interface the same illustrates polymorphism at its lowest level. On a higher level, polymorphism permits extensive code reuse.

Classes and Instances

In REXX, objects are organized into classes. The reason is not unlike why biologists categorize living things in hierarchies--to organize them by their similarities and lines of inheritance. *Classes* are like templates; they define the methods and variables a group of similar objects have in common. Rather than replicate the same methods and variables over and over for each similar object in your program, classes provide an efficient way to store these common actions and characteristics in one place.

If you were writing a program to manipulate some screen icons, for example, you might create an Icon class. In that Icon class you could include all the icon objects with similar actions and characteristics:

A simple class

```
Icon class

OS/2 system icon instance
shredder icon instance
information icon instance
.
.
.
```

All the icon objects might use common methods like DRAW or ERASE. They might contain common variables like position or color or size. What makes each icon object different from one another is the data assigned in its variables. Perhaps for the OS/2 system icon, position='20,20', while for shredder it's '20,30' and for information it's '20,40':

Icon class

```
Icon class

OS/2 system icon instance
  (position='20,20')

shredder icon instance
  (position='20,30')

information icon instance
  (position='20,40')
```

Objects that belong to a class are called *instances* of that class. As instances of the Icon class, the OS/2 system icon, shredder icon, and information icon *acquire* the methods and variables of that class. The instance objects behave exactly as if they each had their own separate methods and variables of the same names. All the instances actually store, however, are their own unique properties--the *data* associated with the variables. Everything else can be retained at the class level.

Instances of the Icon class

```
Icon class
  (position=)

OS/2 system icon instance
  ('20,20')

shredder icon instance
  ('20,30')

information icon instance
  ('20,40')
```

A very useful thing about this arrangement is that, should it become necessary to update or change a particular method, you would only have to change it in one place, at the class level. This single update would then be acquired by every new instance that used the method. In cases where a class has hundreds of instances, the efficiency of this approach is clear.

A class that can create instance objects is called an *object class.* Our hypothetical Icon class is an object class you can use to create other

objects with like properties. In the future you may want to create an application icon or a drives icon.

An object class can produce objects as a cookie cutter produces cookies. It is like a factory for producing objects. In particular, these are a class's *instance* objects.

Data Abstraction

The ability to create new, high-level data types and organize them into a meaningful class structure is called *data abstraction.* Data abstraction is at the heart of object-oriented programming. Once you model objects with real-world properties from the basic data types, you can continue creating, assembling, and combining them into more and more complex objects at will. Then you can use these objects as if they were part of the original programming language. You can do most of your coding in a high-level, real-world context (roll the ball, reverse the pen), instead of the typical low-level, built-in data-type format (using numbers, characters, and symbols) of conventional programming.

Subclasses, Superclasses, and Inheritance

When you write your first object-oriented program, you do not have to begin your real-world modeling from scratch. REXX has done some of this up-front work for you, providing predefined classes and methods. From there you can create additional classes and methods of your own, as you need them.

REXX classes are hierarchical. Any *subclass* (classes below a class in the hierarchy) will *inherit,* or be able to use, the methods and variables of one or more *superclasses* (classes above a class in the hierarchy).

Superclass and subclasses

```
               Superclass


Subclass     Subclass       Subclass
```

You can add a class to an existing superclass. For example, you might add the Icon class to the Screen-Object superclass:

The Screen-Object superclass

```
          Screen-Object class


Icon class   Window class   Bitmap class
```

Doing this lets the subclass inherit additional methods from the superclass. Sometimes a class may have more than one superclass. (Subclass Bitmap might have superclasses Screen-Object *and* Art-Object.) Acquiring methods and variables from more than one superclass is known as *multiple inheritance* .

Multiple inheritance

```
             Screen-Object     Art-Object
```

```
Icon          Window          Bitmap
```

The Basics of Classes

For programming efficiency, similar objects in REXX are grouped into classes, forming a hierarchy. REXX gives you a basic class hierarchy to start with. All of the classes in the hierarchy are described in detail in the *Object REXX Reference*. But briefly, here are the classes REXX provides (there may be others in the system). The classes are indented to show subclassing. Classes in the list that are described later in this section are highlighted in **bold** type:

Object

- **Alarm**
- **Class**
- **Collection classes**
  - Array
  - List
  - Queue
  - Table
    - Set
  - Directory
  - Relation
    - Bag
- **Message**
- Method
- **Monitor**
- **Stem**
- **Stream**
- **String**
- **Supplier**

REXX Classes for Programming

The class framework REXX provides is the starting point for object-oriented programming. Some key classes that you are likely to work with are described in the next sections.

The Alarm Class

The *Alarm class* is for creating objects with timing and notification capability. An alarm object is able send any message to any object at any future time, and until then, you can cancel the alarm.

The Collection Classes

Some potentially very useful classes for programmers are those for defining collections of objects that require manipulation. A *collection* is an object that contains a number of *items* , which can be any objects. These manipulations might include counting objects, organizing them, or assigning them a supplier (for example, to indicate that a specific assortment of baked goods is supplied by the Pie-by-Night Bakery). The classes that enable such manipulations are called *collection classes.*

REXX includes classes for arrays, lists, queues, tables, directories, and the like. Every item stored in a REXX collection has an associated index that you can use to retrieve the item from the collection with the AT or [] (left and right bracket) methods, and each collection defines its own acceptable index types:

Array A sequenced collection of objects ordered by whole number indexes.

List A sequenced collection that lets you add new items at any position in the sequence. A list generates and returns an index value for each item placed in the list. The returned index remains valid until the item is removed from the list.

Queue A sequenced collection with the items ordered as a queue. You can remove items from the head of the queue and add at either its tail or its head. Queues index the items with whole number indexes, in the order in which the items would be removed. The current head of the queue has index 1, the item after the head item has index 2, up to the number of items in the queue.

Table A collection with indexes that can be any object. For example, string objects, array objects, alarm objects, or any user-created object can be a table index. The Table class determines index match by using the == comparison method to test for strict equality. A table contains no duplicate indexes.

Directory
A collection with character string indexes. Index comparisons are performed using the string == comparison method to test for strict equality.

Relation A collection with indexes that can be any object (as with the Table class). A relation can contain duplicate indexes.

Set A collection where the indexes and the values are the same object. Set indexes can be any object (as with the Table class) and each index is unique.

Bag A collection where the index and the value are the same object. Bag indexes may be any object (as with the Table class) and each index may appear more than once.

The Message Class

It is useful to be able to manipulate message objects. For this you use the *Message class*. Methods created for this class are used to send a message, notify the sender object when an error occurs or when message processing is complete, return the results of that processing to the sender or to some other object, process the message object concurrently with the sender object, and so on.

The Monitor Class

The *Monitor class* provides a way to forward messages to a specified destination. Monitor methods let you initialize a monitor object, specify a destination object (or use a previously specified one), and obtain the name of the current destination object.

The Stem Class

A stem is a symbol that must start with a letter and end with a period, like "FRED."or "A.". The value of a stem is a stem object by default. A

stem object is a collection with unique indexes that are character strings. Stem objects are automatically created when a REXX stem variable or REXX compound variable is used. In addition to the items assigned to the collection indexes, a stem object also has a default value that is used for all uninitialized indexes of the collection. You can assign a default value to a stem object and later retrieve this value.

The Stream Class

Input and output streams let REXX communicate with external objects--people, files, queues, serial interfaces, displays, networks, and the like. In programming there are many useful stream actions that could be coded as methods for manipulating the various stream objects. These methods and objects are organized in the *Stream class.*

The methods are used to open streams for reading or writing, close streams at the end of an operation, move the line-read or line-write position within, say, a file stream, or get information about a stream. Methods are also provided to get character strings from a stream or send them to a stream, count characters in a stream, flush buffered data to a stream, query path specifications, time stamps, size, and other information from a stream, or do any other I/O stream manipulation one might find necessary or useful (see Input and Output for examples).

The String Class

Strings are data values that can have any length and contain any characters. The ability to manipulate strings in multiple ways is just as useful as the ability to manipulate I/O streams. There is the need to compare strings: is one string equal to, greater than, or less than another?

Strings are subject to logical operations like AND, OR, exclusive OR, and logical NOT. We need to concatenate and copy strings, reverse them, join them together and split them apart. When strings are numeric, there is the need to perform arithmetic operations on them or find their absolute value or convert them from binary to hex, and vice versa. All this and more can be accomplished using the *String class* of objects.

The Supplier Class

Some collections have suppliers: a bakery, for example, may supply certain breads, cookies, cakes and pies; a financial report may supply certain data, statistics, tables, and reports. The *Supplier class* is used to enumerate items that a collection contained at the time of the supplier's creation. For example, this class contains methods to verify if an item is available from a supplier (Does the Pie-by-Night Bakery sell chocolate cake?). Another method returns the index of the current item in a collection (What is the position of the *apple pie* record?). Others return the current collection item in a collection object, and the next item in the collection.

REXX Classes for Organizing Objects

REXX provides several key classes that form the basis for building class hierarchies. While you may not invoke them in everyday programming, you may want to understand them conceptually.

The Object Class

Because the topmost class in the hierarchy is the *Object class,* everything below it is an object. Objects, to interact with each other, require their own actions, called methods. Those methods that encode actions needed by *all* objects belong to the Object class.

As methods of the top class, every other class in the hierarchy inherits them. For now, think of *inheritance* as the handing down of methods from a "parent" class--called a *superclass* --to all of its "descendent" classes--called *subclasses.* Finally, instances *acquire* methods from their classes. The Object class, then, is the topmost, or root class, of the class hierarchy. Any method created for the Object class is made available to every other class in the hierarchy. From the programmer's standpoint, this happens automatically; it is handled by the REXX language itself.

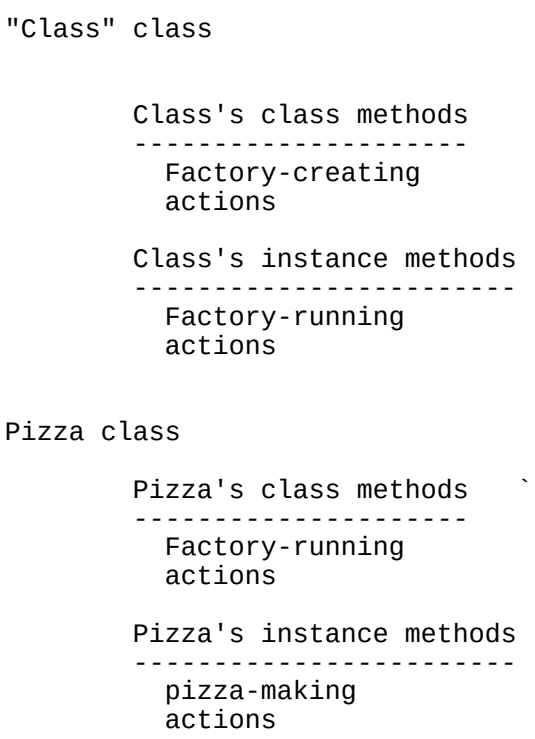
The "Class" Class

Within its class hierarchy REXX includes a class for generating new classes, and this the *Class class.* If a class is like a factory for producing *instances,* Class is like a factory for producing *factories.* Class is the parent of every new class in the hierarchy, and these all inherit Class-like characteristics. The Class-like characteristics take the form of methods and related variables, which reside in Class, to be used by all classes.

A class that can be used to create another class is called a *metaclass.* The Class class is unique among REXX classes in that it is the only metaclass REXX provides (see Metaclasses). As such, Class's methods not only make new classes, they make methods for use by the new class and its instances. They also make methods that only the new class itself can use, apart from its instances. These are called *class methods.* They give a new class some powers that its instances are denied.

Because each instance of Class is another class, or factory, that factory inherits Class's instance methods as class methods. Think of the instance methods of the Class factory as *factory-running* actions. Thus if Class generates a Pizza factory instance, the factory-running actions (Class's *instance* methods) become the *class* methods of the Pizza factory. Factory operations are class methods, and any new methods created to manipulate pizzas would be instance methods:

How subclasses inherit instance methods from the Class class

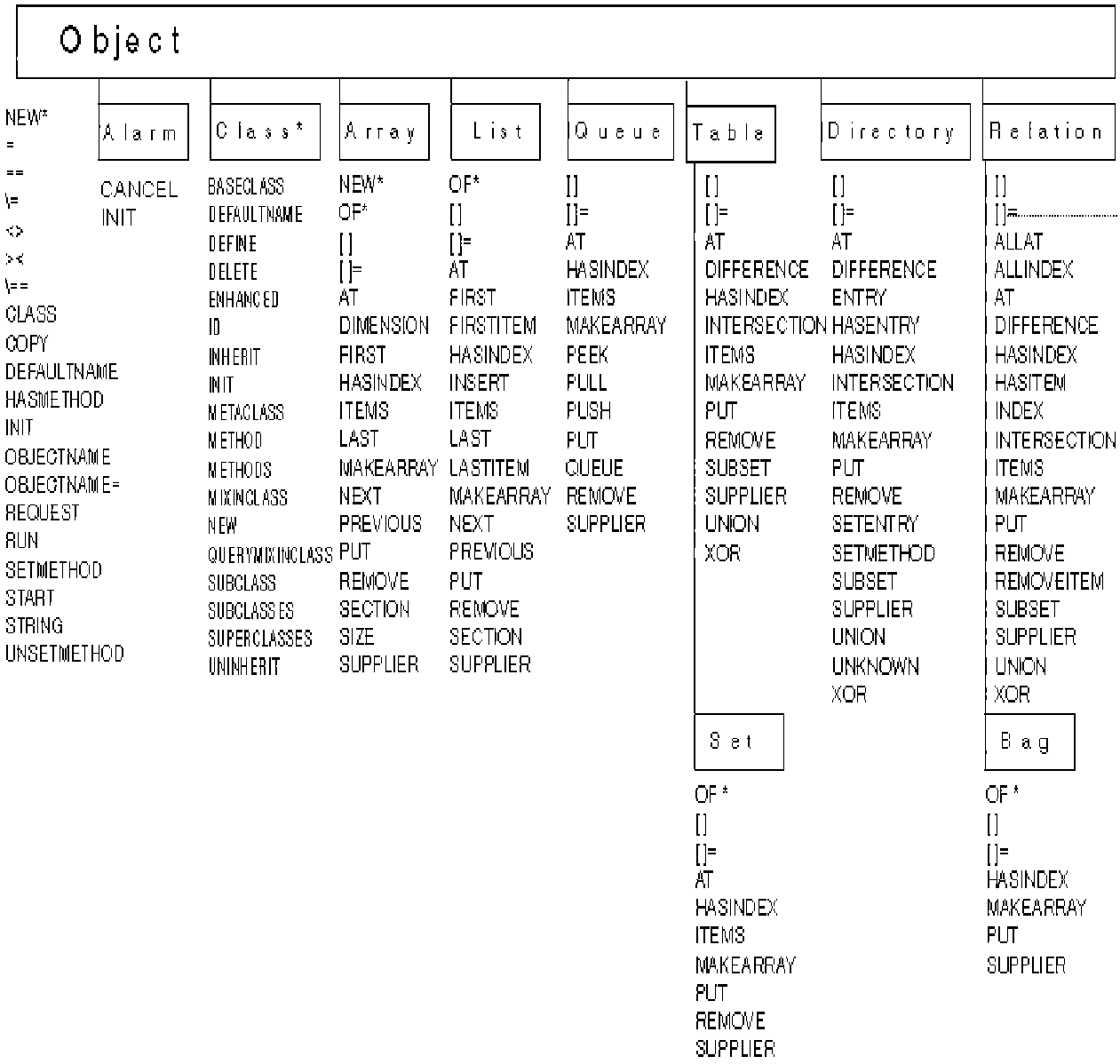


As a programmer, you will typically create classes by using *directives,* rather than the methods of the Class class. In particular, you'll use the ::CLASS directive, described later in this section. The ::CLASS directive is a kind of REXX clause that allows class definitions to be saved permanently, in a file, where they can be reused by other programs. Creating classes "on the fly," by using Class methods sent as messages, is not recommended when permanency or reuse is required. At any rate, directives have class-creating powers similar to the Class methods.

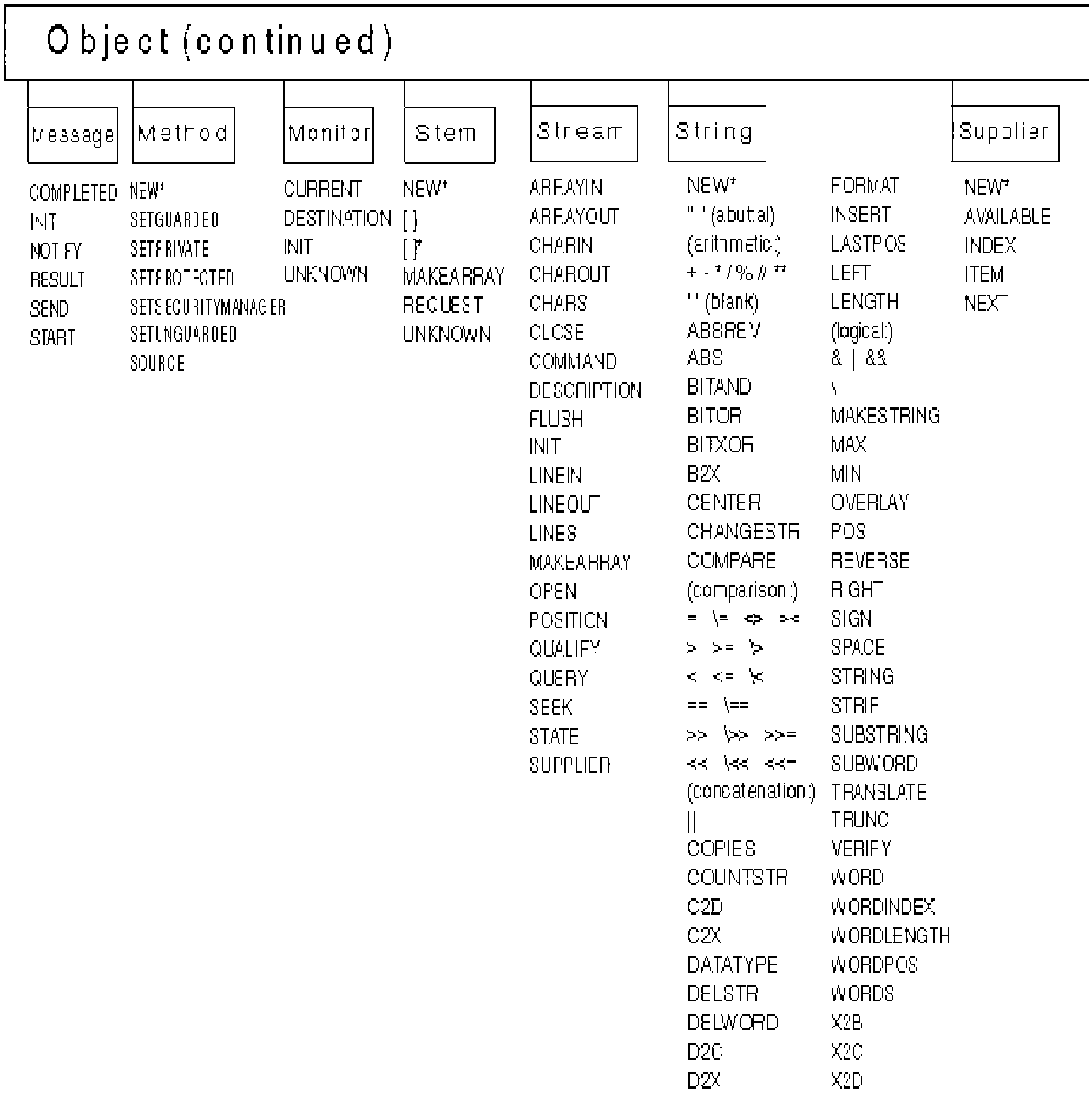
REXX Classes: The Big Picture

The following figure diagrams the supplied REXX classes, along with their methods.

Classes and the inheritance of methods



*All of the methods under the Class class are both class and instance methods. NEW and OF are class methods.



Creating Your Own Classes Using Directives

By analyzing your problem in terms of objects, you can determine what classes need to be created and begin to take advantage of object-oriented technology. You can create a class using messages or directives. Directives are a new kind of REXX clause, and they are preferred over messages because the code is easier to read and understand, especially in large programs. They also provide an easy way for you to save your class definitions and share them with others using the PUBLIC option. Because of these advantages, directives is the technique we will use here.

What Are Directives?

A REXX program is made up of one or more executable units. *Directives* separate these units, which themselves are REXX programs. REXX processes all directives first to set up any classes, methods, or routines needed by the program. Then it runs any code that exists before the first directive. The first directive in a program marks the end of the executable part of the program. A directive is a kind of clause that begins with a double-colon (::) and is non-executable (a directive cannot appear in the expression of an INTERPRET instruction, for example).

The Directives REXX Provides

The following is a short summary of all the REXX directives. See the *Object REXX Reference* for more details on, or examples of, any of these REXX directives.

The ::CLASS Directive

Use the ::CLASS directive to create a class. Programs can then use the new class by specifying it as a REXX environment symbol (the class name preceded by a period) in the program. For example, in A Sample Program Using Directives, the Savings class is created using the ::CLASS directive. Then a program uses the new class by specifying it as an environment symbol, ".savings".

The new class that you create acquires any methods defined by subsequent ::METHOD directives within the program, until either another ::CLASS directive or the end of the program is reached.

You can use the ::CLASS directive's SUBCLASS option to make the new class the subclass of another. In A Sample Program Using Directives, the Savings class is made a subclass of the Account class. A subclass inherits instance and class methods from its specified superclass; in the sample, Savings inherits from Account.

Additional ::CLASS directive options are available for:

- Creating classes using SOM or DSOM information (the EXTERNAL option)
- Inheriting *instance* methods from a specified metaclass as *class* methods of the new class (the METACLASS option); for more on metaclasses, see Metaclasses
- Making the new class available to programs outside its containing REXX program (the PUBLIC option); the outside program must reference the new class by using a ::REQUIRES directive
- Subclassing the new class to a mixin class for purposes of inheriting its instance and class methods (the MIXINCLASS option)
- Adding the instance and class methods of a mixin class to the new class, without subclassing it (the INHERIT option)

When you create a new class, it is always a subclass of an existing class. If you do not specify the SUBCLASS or MIXINCLASS option on the ::CLASS directive, the superclass for the new class is the Object class, and it is not a mixin class.

Your class definition may be in a file of its own, with no executable code preceding it. This is often the case when you are defining classes and methods to be shared by several programs. You put the executable code in another file and refer to the class file using a ::REQUIRES directive.

REXX processes ::CLASS directives in the order they appear, unless there is a dependency on some later directive's processing. Two classes cannot be created having the same class name in one program. If multiple programs contain classes with duplicate names, the last ::CLASS directive processed will be used.

The ::METHOD Directive

After writing a ::CLASS directive, use a ::METHOD directive to create a method for that class and define the method's attributes. The next directive in the program, or the end of the program, will end the method.

Some classes you define will have an INIT method. INIT is called whenever a NEW message is sent to a class. The INIT method should contain whatever code is needed to initialize the object.

While it is not required that a ::METHOD directive be preceded by a ::CLASS directive, without it the method is only accessible to the executable part of the program through REXX's .METHODS environment symbol. This symbol identifies a directory of methods that you can refer to by name. Only one method directive can appear for any method name not associated with a class.

As with the ::CLASS directive, the ::METHOD directive has a number of useful options; in this case they are used for:

- Creating a *class* method for the most-recent ::CLASS directive (the CLASS option)
- Creating a *private* method; that is, a method that works like a subroutine and can only be activated by the object it belongs to--otherwise the method is public by default, and any sender can activate it (the PRIVATE option)
- Creating a method that can be called while other methods are active on the same object, as described in Activating Methods (the UNGUARDED option)
- Creating a pair of instance methods, *method_name* and *method_name*=, for the preceding ::CLASS directive (the ATTRIBUTE option)

The ::ROUTINE Directive

Use the ::ROUTINE directive to create a named routine within a program. The ::ROUTINE directive starts the named routine and another directive (or the end of the program) ends the routine.

The ::ROUTINE directive is useful for defining lower-level routines that are called by several methods. These methods might be in unrelated classes or in different applications. Use ::ROUTINE when you have some useful utility that you don't want to surface as a method.

The ::ROUTINE directive includes a PUBLIC option for making the routine available to programs outside its containing REXX program, and the outside program must reference the routine by using a ::REQUIRES directive.

Only one ::ROUTINE directive can appear for any routine name within a program.

The ::REQUIRES Directive

Use the ::REQUIRES directive when a program needs access to the classes and objects of another program, *program_name.* This directive takes the form:

```
::REQUIRES program_name
```

The ::REQUIRES directive must precede all other directives, and the order of the ::REQUIRES directives determines the search order for the classes and routines defined in the named programs.

REXX uses local routine definitions within a program over routines of the same name accessed through ::REQUIRES directives. Local class definitions within a program override classes of the same name in other programs accessed through ::REQUIRES directives. Another directive (or the end of the program) must follow a ::REQUIRES directive.

How Directives Are Processed

You place any directives (and their method code) *after* all other program code. When you run a program containing directives, REXX:

1. Processes the directives first, to set up the program's classes, methods, and routines.
2. Runs any program code preceding the first directive. This code can use any classes, methods, and routines set up by the directives.

   Once REXX has processed the code preceding the directive, any public classes and objects the program defines are available for other programs (having the appropriate ::REQUIRES directive) to use.

A Sample Program Using Directives

Here is a program that uses directives to create some new classes and methods:

```
asav = .savings~new             /* executable code begins */
say asav~type                   /* executable code        */
asav~name= 'John Smith'         /* executable code ends   */

::class Account                 /* directives begin ...   */

  ::method 'TYPE'
    return "an account"

  ::method 'NAME='
    expose name
    use arg name

::class Savings subclass Account

  ::method 'TYPE'
    return "a savings account"   /* ... directives end     */
```

The preceding program uses the ::CLASS directive to create two classes, the Account class and its Savings subclass. In the `::class Account` expression, the ::CLASS directive precedes name of the new class, Account.

The example program also uses the ::METHOD directive to create TYPE and NAME= methods for Account. In the `::method 'TYPE'` expression, the ::METHOD directive precedes the method name, and is immediately followed by the code for the method. Methods for any new class follow its ::CLASS directive in the program, and precede the next ::CLASS directive.

In the `::method 'NAME='` method, the USE ARG instruction retrieves the argument. The EXPOSE instruction (which must immediately follow the ::METHOD directive) makes the value (here, "John Smith") available for use by other methods. A variable on an EXPOSE instruction is called an *object variable.*

You don't have to do anything (like define an object handle) to associate object variables with a specific object. REXX keeps track of object variables for you. Whenever you send a message to savings account Asav, which points to the Name object, REXX knows what internal object value to use. If you assign another value to Asav (such as "Mary Smith"), REXX will delete the object that was associated with Asav ("John Smith") as part of its normal garbage-collection operations.

In the Savings subclass, a second TYPE method is created that supersedes the TYPE method Savings would otherwise have inherited from Account. Note that the directives appear after all the other program code.

Another Sample Program

We mentioned that a directive is non-executable code that begins with a double-colon (::) and follows all the other program code. The

::CLASS directive creates a class; in this sample, the Dinosaur class. The sample provides two methods for the Dinosaur class, INIT and DIET. These are added to the Dinosaur class using ::METHOD directives. After the line containing the ::METHOD directive, the code for the method is specified. Methods are ended either by the start of the next directive or by the end of the program.

Since directives must follow the executable code in your program, you put that code first. In this case, the executable code creates a new dinosaur, Dino, that is an instance of the Dinosaur class. REXX then runs the INIT method (REXX runs any INIT method automatically whenever the NEW message is received). Here the INIT method is used to identify the type of dinosaur. Next the program runs the DIET method to determine whether the dinosaur eats meat or veggies. The information returned by INIT and DIET is saved by REXX as variables in the Dino object.

In the example, the Dinosaur class and its two methods are defined following the executable program code:

```
dino=.dinosaur~new         /* Create a new dinosaur instance and initialize variables */
dino~diet                  /* Run the DIET method                                      */
exit



::class Dinosaur           /* Create the Dinosaur class  */

  ::method init            /* Create the INIT method     */
    expose type
    say "Enter a type of dinosaur."
    pull type
    return

  ::method diet            /* Create the DIET method     */
    expose type
    select
    when type="T-REX" then string="Meat-eater"
    when type="TYRANNOSAUR" then string="Meat-eater"
    when type="TYRANNOSAURUS REX" then string="Meat-eater"
    when type="DILOPHOSAUR" then string="Meat-eater"
    when type="VELICORAPTOR" then string="Meat-eater"
    when type="RAPTOR" then string="Meat-eater"
    when type="ALLOSAUR" then string="Meat-eater"
    when type="BRONTOSAUR" then string="Plant-eater"
    when type="BRACHIOSAUR" then string="Plant-eater"
    when type="STEGOSAUR" then string="Plant-eater"
    otherwise string="Type of dinosaur or diet unknown"
    end
    say string
    return 0
```

Creating Classes Using Messages

We mentioned that you can create a class using messages as well as directives. Though classes are available only to the program that creates them, there are occasions when this is useful and public availability is not required. Hence we demonstrate the message technique here, using the Savings Account example we showed using directives.

Defining a New Class

To define a new class using messages, you send a SUBCLASS message to the new class's superclass. That is, send the message to the class that will be immediately above the new class in the hierarchy. So, to define a subclass of the Object class called Account, enter:

```
account = .object~subclass('Account')
```

Here, `.object` is a reference to the REXX Object class. `.Object` is an environment symbol indicating the intention to create a new class that is a subclass of the Object class. Environment symbols represent objects in a directory of public objects, called the *Environment object*. These public objects are available to all other objects, and include all the classes that REXX provides. Environment symbols begin with a period and are followed by the class name. Thus the Object class is represented by `.object,` the Alarm class by `.alarm,` the Array class by `.array,` and so on.

The "twiddle" (~) is the "message send" symbol, `subclass` is a method of Class, and the string identifier in parentheses is an argument of SUBCLASS that names the new class, Account.

Adding a Method to a Class

Use the DEFINE method to define methods for your new class. To define a TYPE method and a NAME= method, enter:

```
account~define('TYPE', 'return "an account"')
account~define('NAME=', 'expose name; use arg name')
```

Defining a Subclass of the New Class

Using the SUBCLASS method, you can define a subclass for your new class and then a method for that subclass. To define a Savings subclass for the Account class, and a TYPE method for Savings, enter:

```
savings = account~subclass('Savings Account')
savings~define('TYPE', 'return "a savings account"')
```

Defining an Instance

Use the NEW method to define an instance of the new class, and then call methods that the instance inherited from its superclass. To define an instance of the Savings class named "John Smith," and send John Smith the TYPE and NAME= messages to call the related methods, enter:

```
newaccount = savings~new
say asav~type
asav~name = 'John Smith'
```

Now that you've read how to create your own classes, what *types* of classes might you want to create? The next section explains what you can do using the different types of classes.

Types of Classes

In REXX there are three class types:

- Object classes
- Abstract classes
- Mixin classes

An *object class* (the default) can create instance objects in response to receiving a NEW or ENHANCED message. An *abstract class* serves mainly to organize other classes in the hierarchy and define their message interface, rather than create objects. A *mixin class* , through multiple inheritance, can be designated as an additional superclass to a class. The mixin class typically possesses methods useful to the class that inherits it, but these must be *specifically added* because they lie outside the class's normal line of inheritance.

The following sections explain these class types in more detail.

Object Classes

An object class is like a factory for producing instances. An object class creates instances and provides methods that these instances can use. At the time of its creation, an instance acquires all instance methods of the class it belongs to. If a class adds new methods later, existing instances do not acquire them. Instances created after the new methods do acquire them.

Because classes define methods for their instances, and methods define the variables instances use, object classes are a complete blueprint for creating REXX instances. The Array class is an example of an object class.

Abstract Classes

An *abstract class* defines methods its subclasses can inherit, but typically has no instances. Rather, it serves to organize other classes in the hierarchy. An abstract class can be used to "filter out" a group of shared methods from a number of subclasses, so they do not have to exist in two places.

An abstract class pushes common elements farther up the hierarchy, thus providing a higher level of organization. By filtering out and moving common methods upwards, the abstract class refines the message interface for its subclasses. This lays the groundwork for polymorphism, creating well-defined interfaces for users of the hierarchy. Abstract classes inherit the instance methods of the Class class.

You can create a new abstract class the same way you create an object class. Just use a simple ::CLASS directive; no options are required. While abstract classes are not intended for creating instances, REXX does not prevent you from doing so.

Mixin Classes

The *mixin class* lets you optionally add a set of instance and class methods to one or more other classes using inheritance. You use mixins, therefore, to extend the scope of a class beyond the usual lines of inheritance defined by the hierarchy. Think of it as widening a class's inheritance to accept methods from a sibling or cousin, as well as a parent. When a class inherits from more than just its parent superclass, we call this *multiple inheritance.*

You can add mixin methods to a class by using the INHERIT option on the ::CLASS directive. The class to be inherited must be a mixin class. During class creation and multiple inheritance, subclasses inherit both class and instance methods from their superclasses.

A mixin's first non-mixin superclass is its *base class.* Any subclass of a mixin's base class can directly or indirectly inherit a mixin; other classes cannot.

To create a new mixin class, use the ::CLASS directive with the MIXINCLASS option. A mixin class is also an object class and so can create its own instances.

Metaclasses

A *metaclass* is a class you can use to create another class. REXX provides just one metaclass, the "Class" class. We've said that a class is a factory for creating *instances,* and the "Class" class is a factory for creating *factories.* Whenever you create a new factory, or class, the new class is an instance of Class. The *instance* methods of Class provide the operations needed to run the new factories. These instance methods are inherited by the new factory as its *class* methods.

The classes REXX provides do not permit changes or additions to their method definitions. As a result, all new factories inherit these unchangeable actions from the "Class" class, and thus operate the same way. So if you want to create a new class--a new *factory* --that behaves differently from the others, you can do one of two things:

1. Write additional class methods for the new class, using the ::METHOD directive with the CLASS option
2. Use a metaclass

You can always do 1, and most of the time you will only want to do 1. But if you plan to create many factories with the same operational changes, you might elect to do 2.

Any metaclass you create will be a subclass of the Class class. To make your own metaclass, specify `class` as a SUBCLASS option in the ::CLASS directive:

```
/* Create a new metaclass  */
::class your_metaclass subclass class
```

The instance methods of *your_metaclass* will ultimately become the class methods for any new class created using *your_metaclass.* For example, you could create a metaclass called InstanceCounter that includes instance methods for tracking how many instances the class creates:

```
/* Create a new metaclass that will count its instances        */
::class InstanceCounter subclass class
  ::method init
   .
   .
   .
```

Now, instead of having to add instance-counting class methods to other new classes you write, you can make InstanceCounter their metaclass. When you create the new class, just specify InstanceCounter as a METACLASS option in the ::CLASS directive. If you were creating a Point class, it might look like this:

```
/* Create a public Point class using the InstanceCounter metaclass  */
::class point public metaclass InstanceCounter
  ::method init
   .
   .
   .
```

Instance methods in your new InstanceCounter metaclass will become the class methods of the Point class, and any other classes you create using a similar directive. Here is a complete example:

```
/* a metaclass example */

a = .point~new(1,1)                  /* Create point instances       */
say 'Created point instance' a       /* a, b, and c                  */
b = .point~new(2,2)
say 'Created point instance' b
c = .point~new(3,3)
say 'Created point instance' c

                                     /* Ask the Point class how many */
                                     /* instances it has created     */
say 'The point class has created' .point~instances 'instances.'

                                     /* Create a new metaclass that  */
                                     /* will count its instances     */
::class InstanceCounter subclass class
  ::method init                      /* Create an INIT method to     */
    expose instanceCount             /* initialize instanceCount     */
```

```
    instanceCount = 0                /* Forward INIT to superclass   */

    .message~new(self, .array~of('INIT',super), 'a', arg(1,'A'))~send

  ::method new                       /* Create a NEW instance method */
    expose instanceCount             /* Create a new instance        */
    instanceCount = instanceCount + 1 /* Bump the count              */

                                     /* Forward NEW to superclass    */
    return .message~new(self, .array~of('NEW',super), 'a', arg(1,'A'))~send


  ::method instances                 /* Create an INSTANCES method   */
    expose instanceCount             /* Return the instance count    */
    return instanceCount

                                     /* Create Point class using     */
                                     /* InstanceCounter metaclass    */
::class point public metaclass InstanceCounter
  ::method init                      /* Create an INIT method        */
    expose xVal yVal                 /* Set object variables         */
    use arg xVal, yVal               /* as passed on NEW             */

  ::method string                    /* Create a STRING method       */
    expose xVal yVal                 /* Use object variables         */
    return '('xVal','yVal')'         /* to return string value       */
```

A Closer Look at Objects

Now that you know about classes and how to create them, we'll cover the mechanics of using objects and methods in more detail. First, here is a quick refresher.

We have said that a REXX object consists of:

- Actions coded as *methods*
- Characteristics, coded as *variables,* and their values, sometimes referred to as "state data"

Sending a message to an object causes it to perform a related action. The method whose name matches the message performs the action. The message is the interface to the object, and with *information hiding,* only methods that belong to an object can access its variables.

Objects are grouped hierarchically into *classes.* The class at the very top of the hierarchy is the Object class. Everything below it in the hierarchy belongs to the Object class and is therefore an object. As a result, all classes are objects.

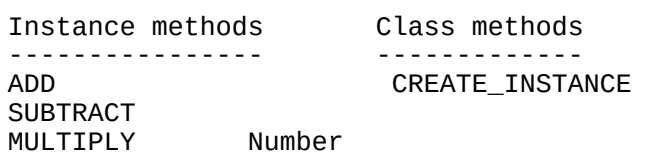
In a class hierarchy, classes, superclasses, and subclasses are relative to one another. Unless designated otherwise, any class directly above a class in the hierarchy is a superclass. And any class below is a subclass.

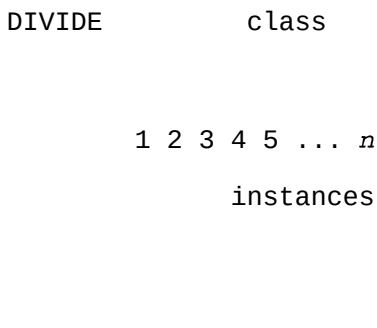
From a class object you can create *instances* of the class. *Instances* are merely similar objects that fit the template of the class; they belong to the class, but are not classes themselves. Instances are the most basic, most elemental of objects.

Both classes and their instances are objects. All objects contain variables and methods, and so this is true for class objects. The methods a class provides for use by its various instances are called *instance methods.* In effect, these define which messages an instance can respond to. Instance methods are by far the most common methods in REXX, and here we simply call them "methods."

The methods available to the class object itself are called *class methods.* (They are actually the instance methods of the Class class.) They define messages that only the class--and not its instances--can respond to. Class methods generally exist to *create* instances and are much less common. When referring to these rarer methods, we will be sure to identify them specifically as "class methods."

Instance methods and class methods





Using Objects in REXX Clauses

To extend our exploration of objects, let's begin with the Myarray object. These examples with Myarray illustrate how to use new objects in REXX clauses. In the expression:

```
myarray=.array~new(5)
```

a new instance of the Array class, Myarray, is created. (Recall that a period precedes a class name in an expression, to distinguish the class from other variables.) Our Myarray array object has five elements.

After Myarray is created, you'll want to assign values to it. One way is with the PUT method. PUT has two arguments, which must be enclosed in parentheses. The first argument is the string to be placed in the element. The second is the number of the element in which to place the string. Here, we associate the string object "Hello" with the third element of Myarray:

```
myarray~put("Hello",3)
```

REXX dynamically adjusts the size of the array to accommodate the new element.

One way to retrieve values from an array object is by sending it an AT message. In the next example, the SAY instruction displays the third element of Myarray:

```
say myarray~at(3)

Results:

Hello
```

The SAY instruction expects the string object as input, and that is precisely what AT returns. So what happens if you put a non-string object in the SAY instruction? SAY sends a STRING message to the object. The STRING method for REXX's built-in objects returns a human-readable string representation for the object. In our example, the STRING method for Myarray returns the string "an array":

```
say myarray  /* SAY sends STRING message to Myarray */

Results:

an array
```

Whenever a method returns a string, you can use it within expressions that require a string. Here, the element of the array that AT returns is a string, so you can put an expression containing the AT method inside a string function like COPIES():

```
say copies(myarray~at(3),4)

Results:
HelloHelloHelloHello
```

This example gets the same result using only methods:

```
say myarray~at(3)~copies(4)
```

Notice that the expression is evaluated from left to right. You can also use parentheses to enforce an order of evaluation.

Almost all messages are sent using the twiddle, but there are exceptions. The exceptions have been made to improve the reliability of the language. This example uses the []= method (left-bracket right-bracket equal-sign) and [] method to set and retrieve array elements:

```
myarray[4]="the fourth element"
say myarray[4]
```

Although the above two statements look like an ordinary array assignment and array reference, they are really messages to the Myarray object. You can prove it to yourself by executing these equivalent statements, which use the twiddle to send the messages.

```
myarray~"[]="("a new test",4)
say myarray~"[]"(4)
```

Similarly, expression operators (such as +,-,/, and *) are actually methods. But you don't have to use the twiddle to send them:

```
say 2+3      /* Displays 5 */
say 2~'+'(3) /* Displays 5 */
```

In the second SAY statement, quotes are needed around the + message because it is a character not allowed in a REXX symbol.

Now let's move on to methods. There are three in particular that you'll use often.

Common Methods You'll Want to Define

When running your program, three methods that REXX looks for, and runs automatically when appropriate, are INIT, UNINIT, and STRING.

Initializing Instances Using INIT

Recall that object classes are those that can create instance objects. When these instances require initialization, you'll want to define an INIT method to set a particular starting value or initiate some startup processing. REXX will look for an INIT method whenever a new object is created; if INIT is there, REXX runs it.

The purpose of initialization is to ensure that the variable is set correctly before using it in an operation. If an INIT method is defined, REXX runs it after creating the instance. Any initialization arguments specified on the NEW or ENHANCED message are passed to the INIT method, which may use them to set the initial state of object variables.

If an instance has more than one INIT method (for example, INIT is defined in multiple classes), each INIT method must forward the INIT message up the hierarchy and run the topmost version of INIT, to properly initialize the instance. An example in the next section demonstrates use of INIT.

Returning String Data Using STRING

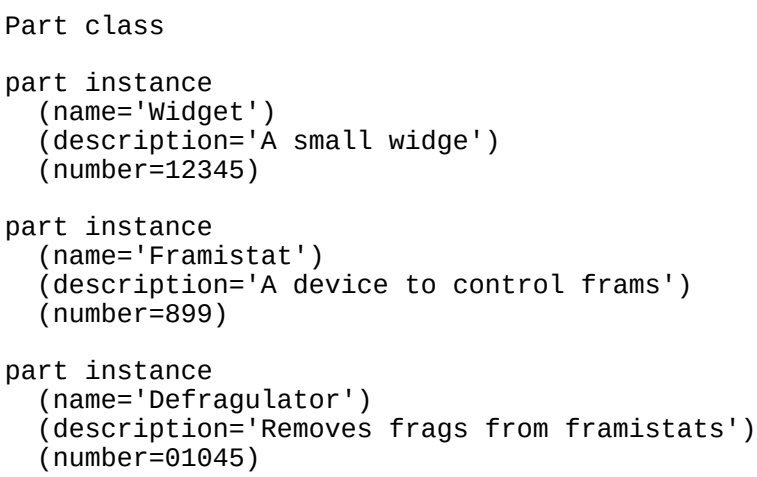
Another useful method is STRING. The STRING method is a handy way to access object data and return it in string form for use by your program. When a SAY instruction is processed in REXX, REXX automatically sends a STRING message to the object specified in the expression. In doing this, REXX uses the STRING method of the Object class and returns a human-readable string representation for the object. For example, if you instruct REXX to `say a,` and `a` is an array object, REXX returns "an array." You can take advantage of this automatic use of STRING by overriding REXX's STRING method with your own, and extract the object data itself--in this case, the actual array data.

Let's look at two programs that demonstrate STRING and INIT. In the first program, the Part class is created, and along with it, the two methods under discussion, STRING and INIT:

```
/* PARTDEF.CMD -- Class and method definition file */

/* Define the Part class as a public class */
::class part public

/* Define the INIT method to initialize object variables */
::method init
expose name description number
use arg name, description, number

/* Define the STRING method to return a string with the part name */
::method string
expose name
return "Part name:" name
```

On the ::CLASS directive, the keyword PUBLIC indicates that the class can be shared with other programs. The two ::METHOD directives define INIT and STRING. Whenever REXX creates a new instance of a class, it calls the INIT method for the class. (We'll see how to create an instance of a class in the next program.) Our INIT method uses an EXPOSE instruction to make the name, description, and number variables available to other methods. These exposed variables are object variables, and are associated with a particular instance of a class.

Instances in the Part class



INIT expects to be passed three arguments. The USE ARG instruction assigns these three arguments to the name, description, and number variables, respectively. Since those variables are exposed, the values are available to other methods.

The STRING method returns the string "Part name: ", followed by the name of a part. The STRING method doesn't expect any arguments. It uses the EXPOSE instruction to tap into the object variables. The RETURN instruction returns the result string.

Now let's see how to use the Part class:

```
/* USEPART.CMD  - use the Part class          */
myparta=.part~new('Widget','A small widge',12345)
mypartb=.part~new('Framistat','Device to control frams',899)
say myparta
say mypartb
exit
::requires partdef
```

The USEPART program creates two "parts", which are instances of the Part class. It then displays the names of the two parts.

REXX processes all directives before running your program. The ::REQUIRES directive indicates that the program needs access to public class definitions that are in another program. In this case, the ::REQUIRES directive refers to the PARTDEF program, which contains the Part definition.

Next, look at the two assignment statements for MypartA and MypartB. These assignment statements create two objects that are instances of the Part class. The objects are created by sending a NEW message to the Part class. The NEW message causes the INIT method to be invoked as part of object creation. Our INIT method takes the three arguments you provide and makes them part of the object's own exclusive set of variables, called a *variable pool.* Each object has its own variable pool (name, description, and number).

The SAY instruction, as we mentioned earlier, sends a STRING message to the object. In the first SAY instruction, the STRING message is sent to MypartA. The STRING method accesses the Name object variable for MypartA and returns it as part of a string. In the second SAY instruction, the STRING message is sent again, but to a different object: MypartB. Because the STRING method is invoked for MypartB, it automatically accesses the variables for MypartB. Recall that you don't need to pass the name of the object to the method in order to distinguish different sets of object variables; REXX keeps track of them for you.

Another way to define classes is by using the SUBCLASS method. You can send a SUBCLASS method to any desired superclass to create a class.

Uninitializing and Deleting Instances Using UNINIT

Object classes, while able to create instances, have no direct control over their deletion. If you assign a new value to a variable, REXX automatically reclaims the storage for the old value in a process called *garbage collection.*

If variables of other objects no longer reference an instance, REXX automatically reclaims that instance. If the instance has allocated other system resources, you must release them at this time using an UNINIT method. REXX cannot automatically release these resources because it is unaware that the instance has allocated them.

In the following example, the value passed to *text* is initialized by REXX using INIT and deleted by REXX using UNINIT. This program makes visible REXX's automatic invocation of INIT and UNINIT by revealing its processing on the screen using the SAY instruction:

```
/* UNINIT.CMD -- example of UNINIT processing */

a=.scratchpad~new('Of all the things I''ve lost')
a=.scratchpad~new('I miss my mind the most')
say 'Exiting program.'
exit

::class scratchpad

  ::method init
    expose text
    use arg text
    say 'Remembering' text

  ::method uninit
    expose text
    say 'Forgetting' text
    return
```

Circumstances dictate when uninitialization processing is needed--when a message object holds an unreported error that should be reported and cleared, for example. If an object requires uninitialization, define an UNINIT method to specify the processing you want.

If UNINIT is defined, REXX runs it before reclaiming the object's storage. If an instance has more than one UNINIT method (for example, UNINIT is defined in multiple classes), each UNINIT method is responsible for sending the UNINIT message up the hierarchy (using the SUPERCLASS overrides) to run the topmost version of INIT.

Special Variables

When writing methods, you can use some special variables available in REXX. A special variable is one that may be set automatically during processing of a REXX program. There are five such variables:

RC is set to the return code from any executed command (including those submitted with the ADDRESS instruction). After the trapping of ERROR or FAILURE conditions, it is also set to the command return code. When the SYNTAX condition is trapped, RC is set to the syntax error number (1-99). RC is unchanged when any other condition is trapped.

Note: Commands executed manually while tracing interactively do not change the value of RC.

RESULT
is set by a RETURN instruction(**) in a subroutine that has been called, or a method that was activated by a message instruction, if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it then RESULT is dropped (becomes uninitialized).

SELF is set when a method is activated. Its value is the object that forms the execution context for the method (that is, the receiver object of the activating message).

You can use SELF to:

- Run a method in an object in which a method is already running. For example, a FIND_CLUES method is running in an object called Mystery_Novel. When FIND_CLUES finds a clue, it sends a READ_LAST_PAGE message to Mystery_Novel:

  ```
  self~read_last_page
  ```

- Pass references about an object to the methods of other objects. For example, a SING method is running in object Song. The code:

  ```
  Singer2~duet(self)
  ```

  would give the DUET method access to the same Song.

SIGL is set to the line number of the last instruction that caused a transfer of control to a label (that is, any SIGNAL, CALL, internal function call, or trapped condition).

SUPER is set when a method is activated. Its value is the class object that is the usual starting point for a superclass method lookup for the SELF object. This is the first immediate superclass of the class that defined the method currently running.

The special variable SUPER lets you call a method in the superclass of an object. For example, the following Savings class has INIT methods that the Savings class, Account class, and Object class define.

```
::class Account

  ::method INIT
    expose balance
    use arg balance
    self~init:super    /* Forwards to the Object INIT method  */

  ::method TYPE
    return "an account"

  ::method name attribute

::class Savings subclass Account

  ::method INIT
    expose interest_rate
    use arg balance, interest_rate
    self~init:super(balance)       /* Forwards to the Account INIT method */

  ::method type
    return "a savings account"
```

When the INIT method of the Savings class is called, the variable SUPER is set to the Account class object. The instruction:

```
self~init:super(balance)
```

calls the INIT method of the Account class rather than recursively calling the INIT method of the Savings class. When the INIT method of the Account class is called, the variable SUPER is assigned to the Object class. Specifying

```
self~init:super
```

calls the INIT method the Object class defines.

You can alter these variables, just like any other variable, but REXX continues to set RC, RESULT, and SIGL automatically when appropriate. EXPOSE, PROCEDURE, USE, and DROP instruction also affect these variables in their usual way.

Certain other information is available to a REXX program. This usually includes the name the program was called by and the source of the program (which are available using the PARSE SOURCE instruction.) In addition, PARSE VERSION makes available the language version and date of REXX implementation that is running. The built-in functions ADDRESS, DIGITS, FUZZ, FORM, and TRACE return other settings that affect the execution of a program.

Public, Local, and Built-in Environment Objects

In addition to the special variables, REXX provides a unique set of objects, called *environment objects.* Environment objects are members of the Object class, and the Object class only. They are like class objects whose class contains only one member, the object itself. REXX makes the following environment objects available:

The Environment Object (.environment)

The Environment object is a directory of public objects that are always accessible. To place something in the Environment directory, use the form:

```
.environment~your.object = value
```

Include a period (.) in any object name you use, to avoid conflicts with current or future REXX entries to the Environment directory. To retrieve your object, use the form:

```
say .environment~your.object
```

To give your object definition limited retrievability, place it in .local, where the scope is the current process only.

Use an *environment symbol* to access the entries of this directory. An environment symbol starts with a period and has at least one other character, and its second character cannot be a digit. You have seen environment symbols earlier; for example in:

```
asav = .savings~new
```

`.Savings` is an environment symbol, and refers to the Savings class. The classes you create can be referenced with an environment symbol, and there is an environment symbol for each REXX-defined class, as well as for each of the unique objects this section describes, such as the NIL object.

The NIL Object (.nil)

The NIL object is a special environment object that does not contain data. It represents the absence of an object, the way a null string represents a string with no characters. Its only methods are those of the Object class. You use the NIL object (rather than the null string) to test for the absence of data in an array entry:

```
if board[row,column] = .nil
then ...
```

All the environment objects REXX provides are single symbols, but use *compound symbols* when you create your own, to avoid conflicts with future REXX-defined entries.

Objects in the Environment object--also called the Global environment object--are available to all programs running on your system. Objects in the Local environment object (below) are available to programs running within the same process.

The Local Environment Object (.local)

The Local environment object is is a directory of process-specific objects that are always accessible. To place something in the Local environment directory, use the form:

```
.local~your.object = value
```

Be sure to include a period (.) in any object name you use, to avoid conflicts with current or future REXX entries to the Local directory. To retrieve your object, use the form:

```
say .local~your.object
```

The scope of .local is the current process. To give your object definition wider retrievability, place it in .environment, where the object will be shared across environments.

Access objects in the Local environment object the same way as in the Environment object. REXX provides the following objects in the Local environment:

.error The Error object (the default error stream) where REXX writes error messages and trace output to.

.input The Input object (the default input stream), which is the source for the PARSE LINEIN instruction, the LINEIN method of the Stream class, and (if you don't specify a stream name) the LINEIN built-in function. It is also the source of the PULL and PARSE PULL instructions if the external data queue is empty.

.output The Output object (the default output stream), which is the destination of output from the SAY instruction, the LINEOUT method (.OUTPUT~LINEOUT), and (if you don't specify a stream name) the LINEOUT built-in function. You can replace this object in the environment to direct such output elsewhere (for example, to a transcript window).

.som The SOM server object.

.somclass
The SOMClass class.

.somclassmgrobject
The SOMClassMgrObject object.

.somobject
The SOMObject class.

Built-In Environment Objects

REXX provides some other environment objects all programs can use. To access these *built-in objects;* you use the special environment symbols whose first character is `.'.

.methods
The .methods environment symbol identifies a directory of methods that ::METHOD directives in the currently running program define. The directory indexes are the method names. The directory values are the method objects. Only methods that are not preceded by a ::CLASS directive are in the .methods directory. If there are no such methods, the .methods symbol has the default value of `.METHODS'. Here's an example:

```
use arg class, methname
class~define(methname,.methods['a'])
::method a
use arg text
say text
```

.rs .rs is set to the return status from any executed command (including those submitted with the ADDRESS instruction). The .rs environment symbol has a value of -1 when a command returns a FAILURE condition, a value of 1 when a command returns an ERROR condition, and a value of 0 when a command indicates successful completion. The value of .rs is also available after trapping of the ERROR or FAILURE condition.

Note: Commands executed manually while tracing interactively do not change the value of .rs. The initial value of .rs is 0.

The Default Search Order for Environment Objects

When you use an environment symbol, REXX performs a series of searches to see if the environment symbol has an assigned value. The search locations and their ordering are:

1. The directory of classes declared on ::CLASS directives within the current program file.
2. The directory of PUBLIC classes declared on ::CLASS directives of other files included with a ::REQUIRES directive.
3. The program local environment directory. The local environment includes process-specific objects such as the .INPUT and .OUTPUT objects. You can directly access the local environment directory by using the .Local environment symbol.
4. The global environment directory. The global environment includes all "permanent" REXX objects such as the REXX supplied classes (.ARRAY and so on) and constants such as .TRUE and .FALSE. You can directly access the global environment by using the .environment environment symbol or using the VALUE built-in function with a null string for the *selector* argument.
5. REXX defined symbols. Other simple environment symbols are reserved for use by REXX for built-in objects. The currently defined built-in objects are .RS and .METHODS.

If an entry is not found for an environment symbol, then the default character string value is used.

Note: You can place entries in both the .local and .environment directories for programs to use. To avoid conflicts with future REXX defined entries, it is recommended that entries you place in either of these directories include at least one period in the entry name.

Example:

```
/* establish a global settings directory */
.local~setentry('MyProgram.settings', .directory~new)
```

Determining the "Scope" of Methods and Variables

Methods interact with variables and their associated data. But a method cannot interact with just any variable. Certain methods and variables are designed to work together. A method designates the variables it wants to work with by exposing them using an EXPOSE instruction. The exposed methods are called object variables. Exposing variables confines them to an object; in object-oriented terms, we say they are *encapsulated.* This protects the object variables' data from being changed by "unauthorized" methods belonging to other objects.

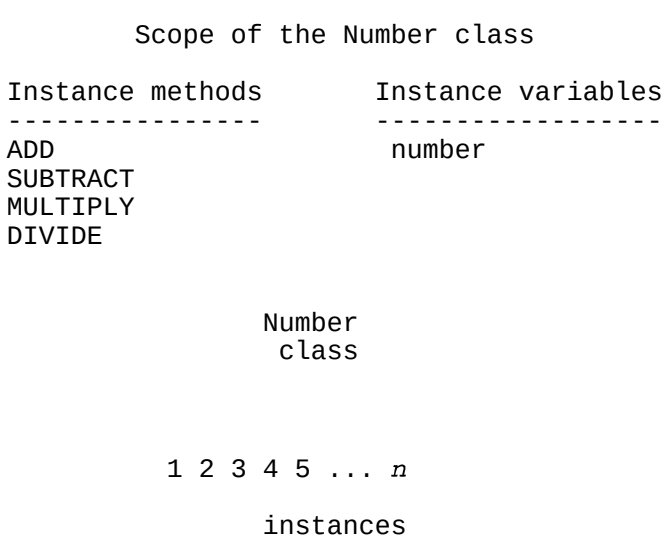
Objects with a Class Scope

Actually, this encapsulation usually takes place at the class level. The class is designed as a template of methods and variables--the common "mold" from which all instances for that class will be stamped. The instances themselves retain only the values of their variables.

Within the hierarchy, then, it is the class structure that ensures the integrity of a class's variables, so that only those methods designed to operate on them, do. (The class structure also provides for easy updating of the method code. If a method requires a change, you only have to change it once, at the class level. The change then is acquired by all the instances sharing the method.)

Associated methods and variables are said to have a certain *scope,* and in this case, that scope is the *class* to which they belong. In other words, the scope of a method and variable group defines or limits the range of objects that can share that group. Typically, a scope consists of the methods and object variables confined to a single class.

Scope of the Number class



Each class in a class hierarchy has a separate scope from every other class. This is what allows a variable in a subclass to have the same name as a variable in a superclass, even though the methods that use the variables may be completely different. But scopes are not confined just to classes.

Objects with Their Own Unique Scope

The methods and variables used by instances in a class are usually found at the class level. But sometimes an instance differs in some way from the others in its class. It may perform an additional action or require some unique handling. In this case one or more methods and related variables can be added directly to the instance. These methods and variables form a totally separate scope, independent of the more-typical class scopes found throughout the rest of the hierarchy.

Methods can be added directly to an instance's collection of object methods using SETMETHOD, a method of the Object class. All subclasses of the Object class inherit SETMETHOD. Alternately, the Class class provides an ENHANCED method that lets you create new instances of a class, whose object methods are the instance methods of its class, but "enhanced" with additional collection methods.

More about Methods

A *method name* can be any character string. When an object receives a message, REXX searches for a method whose name matches the message name.

You need to use quotation marks around a method name when it is the same as an operator. The following example illustrates how to do this correctly. It creates a new class (`Cost`), defines a new method (%), creates an instance of the Cost class (Mycost), and sends a % message to Mycost:

```
mycost = cost~new           /* Creates new instance mycost */
mycost~'%'                  /* Sends % message to mycost   */

::class Cost subclass 'Retail' /* Creates Cost, a sub-     */
                               /* class of 'Retail' class  */
  ::method "%"                 /* Creates % method         */
    expose p                   /* Produces: Enter a price. */
    say "Enter a price"        /* If the user specifies a  */
    pull p                     /* price of 100,            */
    say p*1.07                 /* produces: 107            */
    return 0
```

The Default Search Order for Selecting a Method

When a message is sent to an object, REXX looks for a method whose name exactly matches the message string. If the message is ADD, for example, REXX looks for a method named ADD. Because, in the class hierarchy, there may be more than one method with the same name, REXX begins its search by going first to the object specified in the message. If the sought method is not found there, the search continues up the hierarchy. In order, REXX searches for:

1. A method the object itself defines (with SETMETHOD or ENHANCED)
2. A method the object's class defines

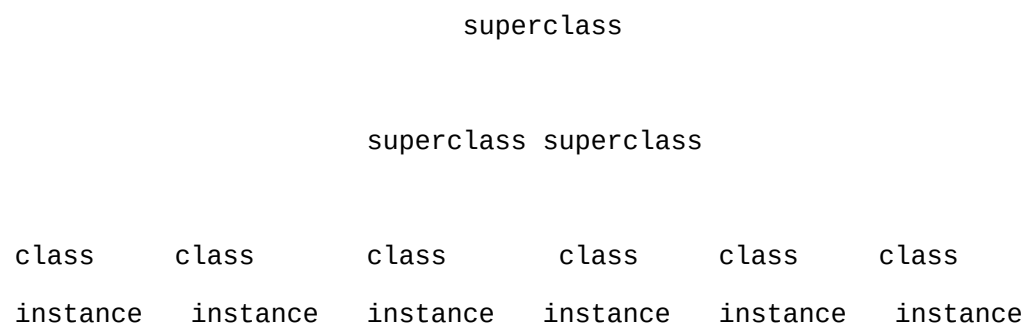
   An object acquires the methods of its parent class; that is, the class for which the object was created. If the class subsequently receives new methods, objects predating the new methods *do not* acquire them.
3. A method an object's superclasses define

   As with the object's class, only methods that existed in the superclass when the object was created are valid. REXX searches the superclass method definitions in the order that INHERIT messages were sent to an object's class.

If REXX doesn't find a match for the message name, REXX checks the object for method name UNKNOWN. If it exists, REXX calls the UNKNOWN method, and returns as the message result any result the UNKNOWN method returns. The UNKNOWN method arguments are the original message name and a REXX array containing the original message arguments (see Defining an UNKNOWN Method). If the object doesn't have an UNKNOWN method, REXX raises a NOMETHOD condition. Any trapped information can then be inspected using REXX's CONDITION built-in function.

REXX searches *up* the hierarchy so that methods existing higher can be supplemented or overridden by methods existing lower.

Searching the hierarchy for a method



```
instance   instance   instance   instance   instance   instance
instance   instance   instance   instance   instance   instance

message
```

For example, suppose you wrote a program that allows users to look up other users' phone numbers. Your program includes a class called Phone_Directory, and all its instances are users' names with phone numbers. You have included a method in Phone_Directory called NOTIFY that reports some data to a file whenever someone looks up a number. All instances of Phone_Directory use the NOTIFY method.

Now you decide you want NOTIFY, in addition to its normal handling, to personally inform you whenever anyone looks up *your* number. To accommodate this special case for your name only, you create your own NOTIFY method that adds the new task and replicates the file handling task. You save the new method as part of your own name instance, retaining the same name, NOTIFY.

Now, when a NOTIFY message is sent to your name instance, the new version of NOTIFY will be found first. REXX will look no further up the class hierarchy. The instance-level version will *override* the version at the class level. This technique of overriding lets you change a method used by one instance without disturbing the common method used by all the other instances. It is very powerful for that reason.

Changing the Search Order for Methods

When composing a message, you can change the default search order for methods by doing both of the following:

1. Making the receiver object the sender object. (You usually do this by specifying the special variable SELF. SELF holds the value of the object in which a method is running. You can use SELF to run another method in an object where a method is already running or pass references about an object to the methods of other objects.)
2. Specifying a colon and a class symbol after the message name. The class symbol identifies the class object to use as the starting point for the search. This class object must be:
   - A direct superclass of the class that defines the active method
   - The object's own class, if you used SETMETHOD to define the active method

   The class symbol is usually the special variable SUPER, but it can be any environment symbol or variable name whose value is a valid class.

In A Sample Program Using Directives, an Account subclass of the Object superclass is created. It defines a TYPE method for Account, and creates the Savings subclass of Account. The example defines a TYPE method for the Savings subclass, as follows:

```
::class Savings subclass Account

  ::method 'TYPE'
    return "a savings account"
```

To change the search order so REXX searches for TYPE in the Account rather than Savings subclass, enter this instead:

```
  ::method 'TYPE'
    return self~type:super '(savings)'
```

When you create an asav instance of the Savings subclass and send a TYPE message to asav:

```
say asav~type
```

REXX displays:

```
an account
```

rather than:

```
a savings account
```

because REXX searches for TYPE in the Account class first.

Public versus Private Methods

A method can be public or private. Any object can send a message that runs a *public* method. Only a message an object sends to itself (using the special variable SELF as the message receiver) can run a *private* method. Private methods include methods at different scopes within the same object. (This lets superclasses make methods available to their subclasses while hiding those methods from clients of the object.) A private method is like an internal subroutine. It shields the internal workings of an object.

Defining an UNKNOWN Method

When an object that receives a message has no matching message name, REXX checks if the object has a method named UNKNOWN. If it does, REXX calls UNKNOWN, passing two arguments. The first is the name of the method that was not located. The second is an array containing the arguments passed with the original message.

To define an UNKNOWN message, specify:

```
UNKNOWN(message_name,message_args)
```

Concurrency

In object-oriented programming, as in the real world, objects interact with each other. This interaction does not usually happen in isolation. Picture, for example, throngs of people interacting at rush hour in the business district of a big city. A program that aspires to simulate the real world would have to enable many objects to interact at any given time. That could mean thousands of objects sending messages to each other, thousands of methods running at once. In REXX, this simultaneous activity is called *concurrency.* To be precise, the concurrency is *object-based* concurrency because it involves objects, as opposed to, say, processes or threads.

REXX objects are inherently concurrent, and this concurrency takes two forms:

- *Inter-object concurrency,* where multiple objects are active--exchanging messages, synchronizing, running their methods--at the same time
- *Intra-object concurrency,* where multiple methods are able to run on the same object at the same time

Default settings in REXX allow full inter-object concurrency but limited intra-object concurrency. Some situations will make it desirable to use special REXX instructions, routines, or classes to make full intra-object concurrency happen. First, let's look at what is automatic.

Full Inter-Object Concurrency

Inter-object concurrency, where objects in a program are actively running methods at the same time, is provided for by REXX in two ways:

- Early reply, by means of the REPLY instruction
- Message objects

Early reply allows the object sending a message to keep on processing after the message is sent. Meanwhile, the receiving object runs the method corresponding to the message. This method contains the REPLY instruction, which returns any results to the sender, interrupting the sender just long enough to reply. The sender and receiver continue operating simultaneously.

Alternatively, an independent *message object* can be created and sent to a receiver. One difference in this approach is that any reply returned does not interrupt the sender. It waits until the sender asks for it. In addition, message objects can simply notify the sender about the completion of the method it sent, and even specify synchronous *or* asynchronous method activation.

The chains of execution represented by the sender method and the receiver method are called *activities*. An activity is a thread of execution that can run methods concurrently with methods on other activities. In other words, activities can run at the same time.

An activity contains a stack of *invocations* that represent the REXX programs running on the activity. An invocation can be:

- A main program invocation
- An internal function or subroutine call
- An external function or subroutine call
- An INTERPRET instruction
- A message invocation

An invocation is pushed onto an activity when an executable unit is invoked. An invocation is removed (or *popped*) when execution completes.

Object Variable Pools

Every object has its own set of variables, called its *object variable pool.* These are variables associated solely with the object--rather than with any one method belonging to the object. When an object's method runs, the first thing it does is identify the object variables it intends to work with. Technically, it "exposes" these variables, using the REXX instruction EXPOSE. Exposing the object's variables distinguishes them from variables used by the method itself, which are not exposed. Every method an object owns--that is, all the *instance methods* in the object's class--can expose variables from the object's variable pool.

Therefore, an object variable pool includes:

- Variables exposed by methods in the object's class (call these variables a *sub* pool)
- Variables inherited from classes elsewhere in the hierarchy (in the form of additional subpools)

Recall that all of a class's variables, together with the methods that expose them, are called a *class scope.* REXX exploits this idea of class scope to achieve concurrency. To explain, the object variable pool is really a collection of variable subpools. Each subpool is at a different scope in the object's inheritance chain. As long as the methods running on the object are at different scopes, they can run simultaneously.

Let's amplify this point further. Scopes, like objects, hide and protect data from outside manipulation. Methods at the same scope share the variable subpools at that scope. The scope shields the variable subpools from methods operating at other scopes. This lets you reuse variable names from class to class, without fear the variables could be accessed and possibly corrupted by a method outside their own class. So class scopes effectively partition an object's variable pool into subpools that can operate independently of one another. Multiple methods can use the same variable pool concurrently, as long as they confine themselves to variables in their own subpools.

Prioritizing Access to Variables

Even with class scopes and subpool partitioning, a variable is still vulnerable if multiple methods within the scope try to access it at the same

time. To handle this, REXX rules that once a particular method is activated and exposes variables from its subpool, that method has exclusive use of the subpool until processing is through. Until then, REXX delays running any other method needing the same subpool.

Thus, within a single scope, if different activities send multiple messages to the same object, REXX forces the methods to run sequentially. This "first-in, first-out" processing of methods in a scope prevents them from simultaneously accessing any one variable, and maybe corrupting the data.

Sending Messages within an Activity

REXX makes one exception to sequential processing--when a method sends a message to itself. Say method M1 has been given exclusive access to object O, and then tries to run a second, *internal* method M2, also belonging to O. Internal method M2 would try to run, but REXX would delay it until original method M1 finished. Yet M1 would be unable to proceed until M2 ran. The two methods would become deadlocked. In actual practice REXX intervenes by treating internal method M2 like a subroutine call. In this case, REXX will run method M2 immediately, then continue processing method M1.

The mechanism controlling this is the activity. Typically, whenever a message is invoked on an object, the activity acquires exclusive access by *locking* the object's scope. Any other activity sending a message to the object whose scope is locked must wait until the first activity releases the lock. The situation is different, however, if the messages originate *from the same activity.* When an invocation running on an activity sends *another* message to the same object, the method is allowed to run because the activity has already acquired the lock for the scope. Thus, REXX permits nested, non-concurrent method invocations on a single activity. No deadlocks occur because REXX treats these additional messages as subroutine calls.

Programming for Intra-Object Concurrency

You've read that multiple methods can access the same object at the same time only if they are operating at different scopes. That is because they are working with separate variable subpools. If two methods in the same scope try to run on the object, REXX by default processes them on a "first-in, first-out" basis, while treating internal methods as subroutines. You can, however, achieve full intra-object concurrency. REXX offers several mechanisms for this, including:

- The SETUNGUARDED method of the Method class and the UNGUARDED option of the ::METHOD directive, which provide unconditional intra-object concurrency.
- The GUARD OFF and GUARD ON instructions, which permit switching between intra-object and default concurrency

When intra-object concurrency at the scope level is needed, you must specifically employ these mechanisms (see the following section). Otherwise, REXX will sequentially process the methods when they are competing for the same object variables.

Activating Methods

By default, REXX assumes that an active method requires exclusive use of its object variable pool. Should another method attempt access at that time, it is locked out until the first method is done with the variable pool. This default intra-object concurrency maintains the integrity of the variable pool and prevents unexpected results. The intra-object concurrency support is especially useful in coding servers. You don't need to manage queues for incoming requests that result in messages being sent to the same object. REXX does it for you.

Some methods *can* function concurrently without affecting the variable pool integrity or yielding unexpected results. When a method does not need exclusive use of its object variable pool, you can use the SETUNGUARDED method or the UNGUARDED option of the ::METHOD directive to provide unconditional intra-object concurrency. These mechanisms control the locking of an object's scope when a method is invoked.

Many methods cannot use SETUNGUARDED and UNGUARDED simply because they require exclusive use of their variable pool some of the time. At other times, they need to perform some action which will involve the concurrent use of the same pool by a method on another

activity. In this case, you can use the GUARD instruction. Using the GUARD instruction, when the method reaches the point in its processing where it requires concurrent use of the variable pool, it invokes GUARD OFF. GUARD OFF lets another method running on a different activity become active on the same object. If the method needs to regain exclusive use, it invokes GUARD ON.

For even more flexibility when activating methods, you can use GUARD ON/OFF with the "WHEN *expression* " option. Add this instruction to the method code at the point where exclusive use of the variable pool becomes conditional. When processing reaches this point, REXX evaluates *expression* to determine if it is true or false.

For example, if you specify "GUARD OFF WHEN *expression* ," the issuing method waits until *expression* becomes true. To become true, another method must assign or drop an object variable that is named in *expression.* Whenever an object variable changes, REXX reevaluates *expression.* If *expression* becomes true, exclusive use of the variable pool is released (because GUARD OFF was specified) and the issuing method resumes with the next instruction.

Computer Karaoke--A REXX Concurrency Sample

Traditional REXX multimedia support provides limited capabilities for synchronizing multimedia events. Although you can use REXX to start a multimedia object, REXX is not notified when the processing is complete. This makes multimedia synchronization an ideal candidate for exploiting concurrency in object-oriented REXX. This sample also uses .alarm and .message environment symbols from the directory of public objects mentioned in the previous section.

The Three Files in This Sample

This sample uses three files. First we'll show you the files, then we'll explain how they work. The first file, CDNOTES.CMD, shows some of the synchronization you can do with REXX:

The CDNOTES.CMD "play and display" file

```
/* CDNOTES.CMD -- plays a CD and displays synchronized text          */
parse arg track                           /* Get a track number     */
if track='' then do
   say 'Must specify a track number...'
   exit
end
script=.script~new('gabriel.'||track)      /* Create script          */
cd=.multimedia~new('cdaudio')              /* Create CDAUDIO object  */
cd~set('time format milliseconds')         /* Set the time format    */
startloc=cd~status('position track' track)  /* Set start point       */
if track=cd~status('number of tracks') then /* Last track?           */
   endloc=''                               /* Set stop point...      */
else
   endloc='to' cd~status('position track' track+1)

/* Play the CD and return control immediately                        */
done=cd~start('play','from' startloc endloc)
script~start                            /* Start synchronized display */
cd~close                                /* Close the CD object        */
rc=.multimedia~free                     /* Free multimedia resources  */
exit

::requires orexxmm
```

CDNOTES plays an audio CD and displays text that is synchronized with the music so that you can sing along. The text is read from a second file, which also contains timing information:

The GABRIEL.7 timing information file

```
0  Digging in the Dirt -- by Peter Gabriel
```

```
25 Something in me...
31 All the time...
37 No way of dealing...
43 Can't go on...
49 This time...
51 This time...
54 This time...
57 I told you...
     .
     .
     .
```

To make this all work, two new classes are needed: one for the text (that is, the script), and one for the multimedia objects. These class definitions are in a third file, OREXXMM.CMD:

The OREXXMM.CMD class definitions file

```
/* Object REXX class and methods for multimedia                    */

::class script public                    /* Class for scripts      */

  ::method init
   expose timeline
   use arg fileid                        /* Get the file ID        */
   file=.stream~new(fileid)              /* Create stream object   */
   timeline=file~makearray('line')       /* Read file into array   */
   file~close                            /* Close the file         */

  ::method start
   expose timeline
   j=timeline~dimension(1)              /* Get the number of lines  */
   do i=1 to j                          /* For each line...         */
                                        /* ...create message object */
     msg=.message~new(self,'do','I',i)  /* ...hurl it into the void */
     .alarm~new(timeline[i]~word(1),msg)
   end                                  /* end do                   */

  ::method do
   expose timeline
   use arg i                            /* Retrieve line number   */
   say time() timeline[i]~subword(2)    /* Display words          */

::class multimedia public         /* Class to "wrap" MCI string     */

  ::method init
   expose object
   use arg object
   /* Initialize the REXX multimedia support                        */
   call rxfuncadd 'mciRxInit', 'MCIAPI', 'mciRxInit'
   call mciRxInit
   /* Open the specified device                                     */
   return mciRxSendString('open' type 'wait','retstr','0','0')

  ::method unknown                /* All MCI string verbs use this  */
   expose object
   use arg mname, margs
   /* Call the MCI support with appropriate verb and arguments      */
   if margs[1]=.nil then margs[1]=''
   return self~mciRxSendString(mname, margs[1] 'wait')

  ::method mciRxSendString
   expose object
   use arg verb, items
   rc=mciRxSendString(verb object items,'retstr','0','0')
   return retstr

  ::method free class            /* Class method to free resources  */
   call mciRxExit
   return result
```

How the Class Definition File Works

Let's start by looking at the class definitions in OREXXMM.CMD.

Three Methods for the Script Class

For the Script class, we provide three methods: INIT, START, and DO. REXX invokes an INIT method for an object whenever a NEW message is sent to the class. For our INIT processing, we read the file containing the script into an array. INIT expects the file identifier as an argument:

```
use arg fileid
```

INIT sends a NEW message to the Stream class to get a new stream object:

```
file=.stream~new(fileid)
```

The Stream class is useful for file I/O in REXX. To read the file, a MAKEARRAY message is sent to the stream object:

```
timeline=file~makearray('line')
```

MAKEARRAY is a very handy method. When used on a stream object, it reads an entire file into an array. After reading the file, INIT sends a CLOSE message to the stream object to close the file.

INIT uses the EXPOSE instruction to make the Timeline array available to all other methods in the Script class.

Next, the START method starts the display of the lines of text. It uses an EXPOSE instruction to access the Timeline array. For each line in the script, START creates a message object. This is the tricky part of the program.

The message object provides for the deferred sending of a message:

```
msg=.message~new(self,'do','I',i)
```

The NEW message causes REXX to run the INIT method of the Message class. The arguments on the NEW message are used by INIT. The first argument on NEW defines where the message should be sent (SELF). The second argument is the message to be sent (DO). The third argument indicates that the next (fourth) argument is an individual string (as opposed to an array). The fourth and final argument is the argument to be passed on the message. The message objects that are created contain the information necessary to send this message:

```
self~do(i)  /* Where "i" is an index to the array of strings */
```

SELF is a special variable. Its value is the object that forms the processing context for the method. In other words, SELF represents the object on which the method is running. In this case, the method that is running is the START method. START runs on an instance of the Script class, so SELF represents the particular instance of the Script class for which START is running.

SELF might seem abstract, but its use becomes immediately apparent when you try to write your own methods. Eventually you'll need to send a message to the object that you are processing. But REXX doesn't pass object handles, so what do you send the message to? You send it to SELF.

After creating a message object, START creates an alarm object for the message. An alarm object sends any message to an object at a given time:

```
.alarm~new(timeline[i]~word(1),msg)
```

The NEW message causes REXX to run the INIT method for the newly-created alarm object. INIT expects a time displacement as its first argument. We pass it the number of seconds into the song that the line in `timeline[i]` is sung. The second argument on INIT is the message object that is to be processed at that time.

After creating all the alarm objects, START returns to its caller. Meanwhile, all the alarm objects are ticking away concurrently, waiting for the correct moments to send the DO messages to SELF. START ends far before all the alarm objects are processed.

The DO method uses EXPOSE to get the Timeline array. DO expects a line number as an argument. These line numbers were put in the message objects when they were created. DO displays the line using a SAY instruction.

Four Methods for the Multimedia Class

For playing the CD, we use the REXX multimedia support currently supplied with OS/2. We want to "wrap" the support with REXX methods so that we can take advantage of REXX concurrency support.

The wrapping involves four methods: INIT, UNKNOWN, MCIRXSENDSTRING, and FREE. The INIT method initializes the REXX multimedia support and opens whatever device is passed to it. CDNOTES will pass `cdaudio' to it so that the audio CD device is opened.

FREE is a class method. We'll use it at the end of CDNOTES to free the multimedia resources.

The most interesting method is UNKNOWN. UNKNOWN is like an "otherwise" clause for a class. If you send an object a message and there isn't a method by that name, the UNKNOWN method is invoked.

In the CDNOTES "play and display" file, we'll be using the MCI string verbs STATUS, SET, PLAY, and CLOSE for the CD. The MCI string command format for those verbs is symmetrical: verb, followed by items, followed by 'WAIT' (required for REXX). We haven't defined these verbs as methods of our Multimedia class. So if we send these verbs as messages to an instance of our Multimedia class, the UNKNOWN method will be invoked.

When REXX invokes the UNKNOWN method, it passes the name of the method as the first argument, and any arguments that were specified as the second argument. This second argument is an array of arguments. Our UNKNOWN method builds an MCI command string from the arguments and then uses the MCIRXSENDSTRING method to process the string.

The MCIRXSENDSTRING method calls the mciRxSendString function and returns RETSTR to the caller.

How the "Play and Display" File Works

Now let's take a look at how CDNOTES uses the classes. CDNOTES reads a script file as input. The script file contains lines of text and the amount of time (in seconds) that must pass before each line is displayed. For our example, we used "Digging in the Dirt" from Peter Gabriel's "Us" CD. We determined the times for each line by playing back the CD using OS/2's CD player, and jotting down the displayed time when each line started.

CDNOTES also has a ::REQUIRES directive for the OREXXMM.CMD, which contains the class definitions for Multimedia and Script.

CDNOTES accepts one argument: a number of a track on the CD to play. It uses the track number as an extension for the file identifier. Thus GABRIEL.7 (for track 7) is used as the file name in the sample.

After retrieving the command line argument, CDNOTES creates a multimedia script by sending a NEW message to the Script class:

```
script=.script~new('gabriel.'||track)  /* Create script      */
```

The name of the file containing the script is passed as an argument on the NEW message.

Next CDNOTES creates an instance of the Multimedia class. The NEW message invokes the INIT method of the Multimedia class. The INIT method expects to be passed the type of multimedia object being opened. In our case, it is CDAUDIO:

```
cd=.multimedia~new('cdaudio')    /* Create CDAUDIO object  */
```

The next few lines of code determine where the end of the track is. The SET and STATUS messages cause REXX to run the UNKNOWN method of the Multimedia class. The UNKNOWN method builds the appropriate MCI string command.

```
cd~set('time format milliseconds')          /* Set the time format  */
startloc=cd~status('position track' track)   /* Set start point      */
if track=cd~status('number of tracks') then  /* Last track?          */
   endloc=''                                 /* Set stop point...    */
else
   endloc='to' cd~status('position track' track+1)
```

Now CDNOTES becomes a bit more interesting. To play the CD, we use the START method of the Object class:

```
done=cd~start('play','from' startloc endloc 'wait')
```

The START method sends a message, which is specified in the arguments, to the CD object. The first argument on the START message is the message to be sent. The second argument on START are the arguments for the message being sent. In this case, the message is PLAY, and the argument that is passed on the PLAY method is the result of the expression:

```
'from' startloc endloc 'wait'
```

The START method sends the message and then immediately returns control to CDNOTES. The PLAY message causes the UNKNOWN method to be processed. UNKNOWN builds an MCI PLAY instruction to start the CD. We didn't send the PLAY message directly to the CD object because control wouldn't return to the CDNOTES program until the CD track finished playing--we wouldn't be able to synchronize the display of the text.

After starting the CD, CDNOTES starts the synchronized text display by sending the START message to the script object:

```
script~start
```

START creates message objects and alarm objects for every line in the song, and returns control immediately to CDNOTES. START is an example of polymorphism. In the Object class, START sends a message and returns immediately. In the Script class, START creates message and alarm objects, and returns immediately. Since the functions are similar, they have the same name.

Notice that we don't account for processing delays that may occur in creating the alarm objects or in starting the audio playback. Consequently, the synchronization won't be perfect. In a real application, you might want to adjust the timings in the script file or add an adjustment factor when creating the alarm objects.

START doesn't wait for the alarm objects to process before returning to its caller. Consequently, START returns to CDNOTES well before the CD track has completed playback. The next line of code closes the CD:

```
cd~close
```

So if the CD is still playing, why doesn't CLOSE stop it prematurely? CLOSE causes REXX to run the UNKNOWN method. The UNKNOWN method runs on the same object (CD) using the same object variables that are in use because of the PLAY method. REXX senses that another method is using the object variable, and blocks the CLOSE until PLAY completes. (You can check this by placing SAY instructions around the line containing the CLOSE.)

After closing the CD, CDNOTES frees multimedia resources by sending the FREE message to the Multimedia class. FREE is a class method.

As this brief sample shows, REXX's object-oriented features greatly extend what you can do with the language. And they do so while maintaining the simplicity and brevity of the traditional REXX. CDNOTES uses objects, concurrent programming, and multimedia to create a simple computer karaoke in about 70 lines of code.

Commands

From a REXX program you can pass commands to OS/2 or to applications designed to work with REXX. When used to run OS/2 commands, REXX becomes a powerful substitute for the OS/2 Batch Facility. You can use variables, control structures, math, and parsing to create procedures that would be impossible to implement with the OS/2 Batch Facility.

Applications that are designed to work with REXX are often referred to as *scriptable* applications. To work with REXX, a scriptable application registers an *environment* with REXX. An environment serves as a kind of workspace shared between REXX and the application. Environments accept application subcommands issued from REXX programs.

For example, many editors provide a command prompt or dialog box from which you can issue subcommands to set margins, add lines, and so on. If the editor is scriptable from REXX, you can issue editor subcommands from a REXX program. These REXX programs are typically

referred to as *macros.* The Enhanced Editor included with OS/2 is an example of an editor that you can script with REXX macros.

When an application runs a REXX macro, REXX directs commands to the application's environment. If you issue a subcommand that the application does not recognize, it may or may not pass the command to OS/2, depending on the application.

To let you specify which environment processes a command, REXX includes an ADDRESS instruction. Starting your REXX programs from the OS/2 command line makes OS/2 the default environment for REXX commands.

How to Issue Commands

REXX makes it easy to issue commands. The basic rule is that whatever REXX cannot process, it passes to the default environment. You can:

- Let REXX evaluate part or all of a clause as an expression. REXX automatically passes the resulting string to OS/2.
- Enclose the entire clause in quotation marks. That renders it a literal string for REXX to pass to OS/2.
- Send a command explicitly to OS/2 by using the ADDRESS instruction.

REXX processes your program one clause at a time. It examines each clause to determine if it is:

- A directive, such as ::CLASS or ::METHOD
- A message instruction, such as:

  ```
  .array~new
  ```

- A keyword instruction, such as:

  ```
  say 'Type total number'
  ```

  or

  ```
  pull input
  ```

- A variable assignment (any valid symbol followed by an equal sign), such as:

  ```
  price = cost * 1.2
  ```

- A label for calling other routines
- A null (empty) clause

If the clause is none of the above, REXX evaluates the *entire clause* as an expression and passes the resulting string to OS/2.

If the string is a valid OS/2 command, OS/2 processes it as though you had typed the string at the command prompt and pressed the Enter key.

The following example shows a REXX clause that uses the DIR command to display a list of files in the current directory.

```
/* display current directory */
say 'DIR command using REXX'
dir
```

The clause `dir` is not a REXX instruction or a label, so REXX evaluates it and passes the resulting string to OS/2. OS/2 recognizes the string DIR as one of its commands and processes it.

Letting REXX evaluate the command as an expression might cause problems, however. Try adding a path to the DIR command in the above program (such as, `dir c:\config.sys`). The OS/2 command in this case is an incorrect REXX expression. The program ends with an error.

A safer way to issue commands is by enclosing the command in quotes, which makes the command a literal string. REXX doesn't evaluate the contents of strings. Because the string isn't a REXX instruction or label, REXX passes the string to OS/2. Here is an example using the PATH command:

```
/* display current path      */
say 'PATH command using REXX'
'path'
```

The next example, DP.CMD, shows a program using the DIR and PATH commands. The PAUSE command is added to wait for the user to press a key before issuing the next instruction or command. Borders are added too.

```
/* DP.CMD -- Issue DIR and PATH commands to OS/2 */

say '='~copies(40)    /* display line of "="'s   */
                      /* for a border            */

'dir'                 /* display listing of      */
                      /* the current directory   */

'pause'               /* pauses processing and   */
                      /* tells user to "Press    */
                      /* any key to continue."   */

say '='~copies(40)    /* display line of '='     */
'path'                /* display the current     */
                      /* PATH setting            */
```

REXX displays the following on the screen when you run the program.

```
[C:\]dp
========================================

 The volume label in drive C is OS2.
 Directory of C:\EXAMPLES

.            <DIR>     10-16-94  12:43p
..           <DIR>     10-16-94  12:43p
EX4_1    CMD     nnnn  10-16-94   1:08p
DEMO     TXT      117  10-16-94   1:10p
     4 File(s)   12163072 bytes free
Press any key when ready . . .

========================================
PATH=C:\OS2;C:\OS2\UTIL;C:\OS2\INSTALL
[C:\]
```

Suppressing the Command Echo

When your REXX program issues an OS/2 command, REXX passes the command to the OS/2 command handler for processing. This processing includes displaying, or *echoing,* the command on the screen. Commands are also echoed in old-style (non-REXX) .CMD files.

To suppress command echoes, issue the ECHO OFF command from your REXX program. OS/2 will suppress all echoing for the remainder of your REXX program. To suppress echoing for one command, put an at sign (@) in front of the command:

```
/* NOECHO.CMD -- suppress command echoes             */
say 'Without echo:'
'@dir c:\config.sys'
say 'With echo:'
'dir c:\config.sys'
say 'Suppress remaining echoes...'
'@echo off'          /* suppress the echo of ECHO OFF */
'dir c:\config.sys'
```

REXX and Batch Files

You can use a REXX program anywhere you now use OS/2 batch files. The following example shows an OS/2 batch file that processes user input to display a help message:

```
@echo off
if %1.==. goto msg
if %1 == on goto yes
if %1 == off goto no
if %1 == ON goto yes
if %1 == OFF goto no
if %1 == On goto yes
if %1 == oN goto yes
if %1 == OFf goto no
if %1 == OfF goto no
if %1 == Off goto no
if %1 == oFF goto no
if %1 == oFf goto no
if %1 == ofF goto no
helpmsg %1
goto exit
:msg
helpmsg
goto exit
:yes
prompt $i[$p]
goto exit
:no
cls
prompt
:exit
```

Here is the equivalent program in REXX:

```
/* HELP.CMD -- Get help for a system message */
arg action .
select
  when action=''    then     'helpmsg'
  when action='ON'  then     'prompt $i[$p]'
  when action='OFF' then do
                           'cls'
                           'prompt'
                           end
  otherwise 'helpmsg' action
  end
exit
```

Issuing a Command to Call a .CMD File

If you are issuing a command to have OS/2 run one of its built-in commands or other programs, you can call it by name as previous examples showed. However, to have OS/2 run another .CMD program from your REXX program, you must call it using a CALL instruction instead of calling it by name. There are two kinds of calls you can use:

- The REXX CALL instruction
- The OS/2 CALL command

The REXX CALL instruction calls other REXX programs. To call a REXX program named MYSUB1, you could write the CALL instruction:

```
call mysub1
```

REXX recognizes the CALL instruction, handles the call, and processes MYSUB1 as a REXX program.

The REXX CALL instruction does not work to call a non-REXX .CMD file. Instead, you would use the OS/2 CALL command. To call a non-REXX .CMD program named MYSUB2, you could write the CALL instruction like this:

```
"call mysub2"
```

REXX evaluates the expression and passes it to the OS/2 command handler for processing. The command handler recognizes the CALL command and processes MYSUB2 as a .CMD program.

The quotation marks around `call mysub2` indicate that this is the OS/2 CALL command instead of a REXX CALL instruction.

Using Variables to Build Commands

You can use variables to build commands, just as you would use variables to construct any string. The SHOFIL.CMD program below is an example. SHOFIL types a file that the user specifies. It prompts the user to enter a file name and then builds a variable to hold the TYPE command and the input file name.

To have REXX issue the command to the operating system, put the variable holding the command string on a line by itself. REXX evaluates the variable and passes the resultant string to OS/2:

```
/* SHOFIL.CMD -- build command with variables */

/* prompt the user for a file name            */
say "Type a file name:"

/* assign the response to variable FILENAME   */
pull filename

/* build a command string by concatenation    */
commandstr = "TYPE" filename

/* Assuming the user typed "demo.txt,"        */
/* the variable COMMANDSTR contains           */
/* the string "TYPE DEMO.TXT" and so...       */

commandstr           /* ...REXX passes the    */
                     /* string on to OS/2     */
```

REXX displays the following on the screen when you run the program.

```
[C:\]shofil
Type a file name:
demo.txt

This is a sample text file. Its sole
purpose is to demonstrate how
commands can be issued from REXX
programs.

[C:\]
```

Using Quotation Marks

The rules for forming a command from an expression are exactly the same as those for forming expressions. Be careful of symbols that

have meanings for both REXX and OS/2 programs. The DIRREX.CMD program below shows how REXX evaluates a command when the command name and a variable name are the same:

```
/* DIRREX.CMD -- assign a value to the symbol DIR */
say "DIR command using REXX"
dir = "echo This is not a directory."

/* pass the evaluated variable to OS/2           */
dir
```

Because Dir is a variable that contains a string, the string is passed to OS/2. The OS/2 DIR command is not executed. Here are the results:

```
[C:\]dirrex
DIR command using REXX:
This is not a directory.
[C:\]
```

REXX evaluates a literal string--a string enclosed in matching quotation marks--exactly as it is. To ensure that a symbol in a command is not evaluated as a variable, enclose it in matching quotation marks as follows:

```
/* assign a value to the symbol DIR       */
say "DIR command using REXX"
dir = "echo This is another string now."

/* pass the literal string "dir" to OS/2  */
"dir"
```

REXX displays a directory listing.

The best way to ensure that REXX passes a string to the OS/2 as a command is to enclose the entire clause in quotation marks. This is especially important when you use symbols that REXX uses as operators.

If you want to use a variable in the command string, leave the variable outside the quotation marks. For example:

```
extension = "BAK"
"delete *."¦¦extension

option = "/w"
"dir"¦¦option
```

ADDRESS Instruction

To send a command to a specific environment, use this format of the ADDRESS instruction:

```
ADDRESS environment expression
```

For *environment* , specify the destination of the command. To address the OS/2 environment, use the symbol CMD. For *expression* , specify an expression that results in a string that REXX passes to the environment. Here are some examples:

```
address CMD "dir"     /* pass the literal string   */
                      /* "dir" to OS/2 program     */

cmdstr = 'dir *.txt'  /* assign a string           */
                      /* to a variable             */

address CMD cmdstr    /* REXX passes the string    */
                      /* "dir *.txt" to OS/2       */
address edit "regen"  /* REXX passes the "regen"   */
                      /* command to a fictitious   */
                      /* environment named edit    */
```

Notice that the ADDRESS instruction lets a single REXX program issue commands to two or more environments.

Using Return Codes from Commands

Information passes between REXX and the environment. REXX sends command strings to the environment, and the environment can return information. One piece of information that the environment passes to REXX is the return code from the command.

With each command it processes, OS/2 produces a number called a *return code.* When a REXX program is running, this return code is automatically assigned to a special built-in REXX variable named RC.

If the command was processed with no problems, the return code is nearly always 0. If something goes wrong, the return code issued is another, nonzero number. What the number is depends on the command itself and the error encountered.

This example shows how to display a return code:

```
/* GETRC.CMD report */
'TYPE nosuch.fil'
say 'the return code is' RC
```

The special variable RC can be used in expressions just like any other variable. In the next example, an error message is displayed when a the TYPE command returns a nonzero value in RC:

```
/* Simple if/then error-handler */
say "Type a file name:"
pull filename
"TYPE" filename
if RC \= 0
then say "Could not find" filename
```

This program tells you only that OS/2 could not find a nonexistent file. This program does not do much more than OS/2 does, but you have the basic idea of how to capture a return code.

A system error alone does not stop the running of a REXX program. Without some provision to stop the program, in this case a *trap* , REXX continues running. You may have to press the Control (Ctrl)+Break keys to stop processing. REXX includes the following instructions for trapping and controlling system errors:

- CALL ON ERROR
- CALL ON FAILURE
- SIGNAL ON ERROR
- SIGNAL ON FAILURE

Subcommand Processing

REXX programs can issue commands or subcommands to programs other than OS/2. To determine what subcommands you can issue, refer to the documentation for the application.

To make your own applications scriptable from REXX, see REXX Application Programming Interface.

Trapping Command Errors

The most efficient way to detect errors from commands is by creating *condition traps,* using the SIGNAL ON and CALL ON instructions, with either the ERROR or the FAILURE condition. When used in a program, these instructions enable, or switch on, a detector in REXX that tests the result of every command. Then, if a command signals an error, REXX stops usual program processing, searches the program for the appropriate label (ERROR:, or FAILURE:, or a label that you created), and resumes processing there.

SIGNAL ON and CALL ON also tell REXX to store the line number (in the REXX program) of the command instruction that triggered the condition. REXX assigns that line number to the special variable SIGL. Your program can get even more information about what caused the command error through the built-in function CONDITION.

Using the SIGNAL and CALL instructions to handle errors has several advantages; namely, programs:

- Are easier to read because you can confine error-trapping to a single, common routine
- Are more flexible because they can respond to errors by clause (SIGL), by return code (RC), or by other information (CONDITION method or built-in function)
- Can catch problems and react to them before the environment issues an error message
- Are easier to correct because you can turn the traps on and off (SIGNAL OFF and CALL OFF)

For other conditions that may be detected using SIGNAL ON and CALL ON, see the *Object REXX Reference.*

Instructions and Conditions

The instructions to set a trap for errors are SIGNAL and CALL. Example formats are:

```
SIGNAL ON condition NAME trapname
CALL   ON condition NAME trapname
```

The SIGNAL ON instruction initiates an exit subroutine that ends the program. The CALL ON instruction initiates a subroutine that returns processing to the clause immediately following the CALL ON instruction. Use CALL ON when you want to *recover* from a command error or failure.

The two command conditions that can be trapped are:

ERROR Detects *any* nonzero error code the default environment issues as the result of a REXX command

FAILURE
: Detects a severe error, preventing the system from processing the command

A failure, in this sense, is a particular category of error. If you use SIGNAL ON or CALL ON to set a trap only for ERROR conditions, then it traps failures as well as other errors. If you also specify a FAILURE condition, then the ERROR trap ignores failures.

With both the SIGNAL and CALL instructions, you have the option of specifying the name of the trap routine. Add a NAME keyword followed by the name of the subroutine. If you don't specify the name of the trap routine, REXX uses the value of *condition* as the name (REXX looks for the label ERROR:, FAILURE:, and so on).

For more information about other conditions that can be trapped, see the *Object REXX Reference.*

Disabling Traps

To turn off a trap for any part of a program, use the SIGNAL or CALL instructions with the OFF keyword, such as:

```
SIGNAL OFF ERROR
SIGNAL OFF FAILURE
CALL OFF ERROR
```

```
CALL OFF FAILURE
```

Using SIGNAL ON ERROR

The following example shows how a program might use SIGNAL ON to trap a command error in a program that copies a file. In this example, an error occurs because the name of a nonexistent file is stored in the variable File1. Processing jumps to the clause following the label ERROR:

```
/* example of error trap                                  */
signal on error                 /* Set the trap.          */
     .
     .
     .
"COPY"  file1 file2             /* When an error occurs... */
     .
     .
exit .
error:                          /* ...REXX jumps to here   */
say "Error" rc "at line" sigl
say "Program cannot continue."
exit                            /* and ends the program.   */
```

Using CALL ON ERROR

If there were a way to recover, such as by typing another file name, you could use CALL ON to recover and resume processing:

```
/* example of error recovery */
call on error
     .
     .
     .
"COPY"  file1 file2
say "Using" file2 `
     .
     .
      exit
error:
say "Can not find" file1
say "Type Y to continue anyway."
pull ans
if ans = "Y" then
   do
      /* create dummy file */
           .
           .
           .
      file2 = "dummy.fil"
      RETURN
```

```
        end
    else exit
```

A Common Error-handling Routine

The following example shows a simple error trap that you can use in many programs:

```
/* Here is a sample "main program" with an error           */
signal on error        /* enable error handling            */
'ersae myfiles.*'      /* mis-typed 'erase' instruction    */
exit

/* And here is a fairly generic error-handler for this     */
/* program (and many others...)                            */
error:
  say 'error' rc 'in system call.'
  say
  say 'line number =' sigl
  say 'instruction = '  ¦¦ sourceline(sigl)
  exit
```

Input and Output

Object REXX supports a stream I/O model. In the stream model, your program reads data from various devices (such as hard disks, CD-ROMs, and keyboards) as a continuous stream of characters. Your program also writes data as a continuous stream of characters.

In the stream model, a text file is represented as a stream of characters with special new-line characters marking the end of each "line" of text in the stream. A binary file is simply a stream of characters without an inherent line structure. REXX lets you read streams as lines *or* as characters.

To support the stream model, Object REXX includes a Stream class and many methods to use on stream objects. To input or output data, you first create an instance of the Stream class that represents the device or file you want to use. For example, the following clause creates a stream object for the file C:\CONFIG.SYS:

```
/* Create a stream object for CONFIG.SYS */
file=.stream~new('c:\config.sys')
```

Then you send the stream object messages that are appropriate for the device or data. CONFIG.SYS is a text file, so you would normally use methods that read or write data as lines. Some of these methods are LINES, LINEIN, and LINEOUT.

If the stream represented a binary file (such as a WAV, GIF, TIF, AVI, or EXE file), you would use methods that read and write data as characters. Some of these methods are CHARS, CHARIN, and CHAROUT.

The Stream class includes other methods for opening and closing streams, flushing buffers, seeking, retrieving stream status, and other input/output operations.

Many of the methods of the Stream class are also available as REXX built-in functions. While you can use the functions, using the Stream class is preferred. In any case, you should not intermix the use of functions and methods *on the same stream.* Doing so yields unpredictable results.

More about Stream Objects

To use streams in Object REXX, you create new instances of the Stream class. These "stream objects" represent the various data sources and destinations available to your program, such as hard disks, CD-ROMs. keyboards, displays, printers, serial interfaces, networks, and so on. Because these sources are represented as objects, you can work with them in similar (but not identical) ways.

Stream objects can be "transient" or "persistent." An example of a transient (or dynamic) stream object is a serial interface. Data can be sent or received from serial interfaces, but the data is not "stored" permanently by the serial interface itself. Consequently, you cannot, for example, seek to a position in the data stream. Once you read or write it, the data cannot be read again.

A disk file is an example of a persistent stream object. Because the data is stored on disk, you can seek forward and backwards in the stream and read data that you have previously read. REXX maintains separate read and write pointers to a stream. You can move the pointers independently using arguments on methods such as LINEIN, LINEOUT, CHARIN, and CHAROUT. REXX also provides SEEK and POSITION methods for setting the read and write positions. (We'll discuss read and write positioning in a later section.)

Reading a Text File

Let's look at an example of reading a file. The following program, COUNT.CMD, counts the words in a text file. To run the program, enter COUNT followed by the name of the file to be processed:

```
count myfile.txt
count r:\rexx\articles\devcon7.scr
```

COUNT uses the String method WORDS to count the words, so COUNT actually counts blank-delimited tokens:

```
/* COUNT.CMD -- counts the words in a file                        */
parse arg path              /* Get file name from command line     */
count=0                     /* Initialize a counter                */
file=.stream~new(path)      /* Create a stream object for the file */
do while file~lines<>0      /* Loop while there are lines          */
  text=file~linein          /* Read a line from the file           */
  count=count+(text~words)  /* Count words and add to counter      */
end
say count                   /* Display the count                   */
```

To read a file, COUNT first creates a stream object for the file by sending the NEW message to the Stream class. The file name (with or without a path) is specified as an argument on the NEW method.

Within the DO loop, COUNT reads the lines of the file by sending LINEIN messages to the stream object (pointed to by the variable File). The first LINEIN message causes REXX to open the file (the NEW method doesn't open the file). LINEIN, by default, reads one line from the file from the current read position.

REXX returns only the text of the line to your program. REXX does not return new-line characters.

The DO loop is controlled by the expression "file~lines<>0." The LINES method returns the number of lines remaining to be read in the file, so REXX processes the loop until no lines remain to be read.

In the COUNT program, the LINEIN request forced REXX to open the file, but you can also open the file yourself using the OPEN method of the Stream class. By using the OPEN method, you control the mode in which REXX opens the file. When REXX implicitly opens a file because of a LINEIN request, it tries to open the file for both reading and writing. If that fails, it opens the file for reading. To ensure that the file is opened only for reading, you could modify COUNT as follows:

```
/* COUNT.CMD -- counts the words in a file                        */
parse arg path              /* Get file name from command line     */
count=0                     /* Initialize a counter                */
file=.stream~new(path)      /* Create a stream object for the file */
openrc=file~open('read')    /* Open the file for reading           */
if openrc<>'READY:' then do /* Check the return code               */
   say 'Could not open' path||'. RC='||openrc
   exit openrc              /* Bail out                            */
end
do while file~lines<>0      /* Loop while there are lines          */
  text=file~linein          /* Read a line from the file           */
  count=count+(text~words)  /* Count words and add to counter      */
end
file~close                  /* Close the file                      */
say count                   /* Display the count                   */
```

The CLOSE method, used near the end of the above example, closes the file. A CLOSE isn't required. REXX will close the stream for you when the program ends. However, it's a good idea to CLOSE streams when you are done with them so that the resource is immediately available for other uses (perhaps by other OS/2 processes).

Reading a Text File into an Array

REXX provides a Stream method, named MAKEARRAY, that reads the contents of a stream into an array object. MAKEARRAY is very handy when you need to read an entire file into memory for processing. You can read the entire file with a single REXX clause--no looping is necessary.

The following example (CVIEW.CMD) uses the MAKEARRAY method to read the entire CONFIG.SYS file into an array object. CVIEW displays selected lines from CONFIG.SYS. A search argument can be specified when starting CVIEW:

```
cview libpath
```

CVIEW will prompt for a search argument if you don't specify one.

If CVIEW finds the string, it displays the line on which the string is found. CVIEW continues to prompt for new search strings until you enter "Q" in response to the prompt.

```
/* CVIEW -- display lines from CONFIG.SYS                          */
parse upper arg search_string      /* Get any command line argument  */
file=.stream~new('c:\config.sys')  /* Create stream object */
lines=file~makearray(line)         /* Read file into an array object */
                                   /* LINES points to the array obj. */
do forever
   if search_string='' then do     /* Prompt for user input          */
      say 'Enter a search string or Q to quit:'
      parse upper pull search_string
      if search_string='Q' then exit
   end  /* Do */
   do i over lines                  /* Scan the array                */
      if pos(search_string,translate(i))>0 then do
         say i                     /* Display any line that matches */
         say '='~copies(20)
      end  /* Do */
   end /* do */
   search_string=''                 /* Reset for next search         */
end /* do */
```

Reading Specific Lines of a Text File

You can read a specific line of a text file by entering a line number as an argument on the LINEIN method. In this example, line 3 is read from CONFIG.SYS:

```
/* Read and display line 3 of CONFIG.SYS */
infile=.stream~new('c:\config.sys')
say infile~linein(3)
```

You don't necessarily reduce file I/O by using specific line numbers. Because text files typically do not have a specific record length, REXX must read through the file counting line-end characters to find the line you want.

Writing a Text File

To write lines of text to a file, use the LINEOUT method. By default, LINEOUT appends to an existing file. The following example adds an item to a to-do list that is maintained as a simple text file:

```
/* TODO.CMD -- add to a todo list                              */
parse arg text
file=.stream~new('todo.dat')      /* Create a stream object    */
file~lineout(date() time() text)  /* Append a line to the file */
exit
```

In TODO.CMD, a text string is provided as the only argument on LINEOUT. REXX writes the line of text to the file and then writes a new-line character. You do not have to provide a new-line character in the string being written.

If you want to overwrite a file, specify a line number as a second argument. When a line number is specified, REXX uses it to position the write pointer before writing:

```
file~lineout('13760-0006',35)  /* Replace line 35 */
```

For text files, write positioning has a somewhat specialized use. REXX does not prevent you from overwriting existing new-line characters in the file. REXX writes whatever line you give it at whatever write position you tell it to use. Consequently, if you want to replace a line of the file without overlaying following lines, *the line you write must be exactly the same length as the line you are replacing.* Writing a line that's shorter than an existing line will leave part of the old line in the file.

Also, don't assume that positioning the write pointer to line 1 will cause the file to be replaced. Yes, REXX will start writing over the existing data starting at line 1, but if you happen to write fewer bytes than previously existed in the file, your data will be followed by the remainder of the old file.

To replace a file, use the OPEN method with WRITE REPLACE or BOTH REPLACE as an argument. In the following example, a file named TEMP.DAT is replaced with a random number of lines. TEMP.DAT is then read and displayed. You can run the example repeatedly to verify that TEMP.DAT is replaced on each run.

```
/* REPFILE.CMD -- demonstrates file replacement                    */
testfile=.stream~new('temp.dat') /* Create a new stream object     */
testfile~open('both replace')    /* Open for read, write, and replace */
numlines=random(1,100)           /* Pick a number from 1 to 100    */
runid=random(1,9999)             /* Pick a run identifier          */
do i=1 to numlines               /* Write the lines                */
   testfile~lineout('Run ID:'||runid 'Line number' i)
end

/* Now read and display the file.  The read pointer is already at the */
/* beginning of the file.  MAKEARRAY reads from the read position to  */
/* the end of the file and returns an array object containing the     */
/* lines.                                                             */
filedata=testfile~makearray('line')
do i over filedata
   say i
end
testfile~close
```

The REPFILE example also demonstrates that REXX maintains separate read and write pointers to a stream. Notice that we started reading without repositioning the read pointer. Since we hadn't yet read anything, the read pointer was still at the beginning of the file. The write pointer, however, is at the end of the file.

Reading Binary Files

For our purposes, a binary file is any file whose data is not organized into lines by new-line characters. In most cases, you'll use the character I/O methods (such as CHARS, CHARIN, CHAROUT) on these files. There is one exception, however, which will also be discussed.

Let's start by reading an entire binary file into a single REXX variable. Suppose, for example, that you feel compelled to read the data in the DOINK.WAV file (supplied with OS/2 multimedia support in c:\mmos2\sounds) into a variable. Here's how to do it:

```
/* GETDOINK -- reads DOINK.WAV into a variable          */
doinkf=.stream~new('c:\mmos2\sounds\doink.wav')
```

```
say 'Number of characters in the file=' doinkf~chars

/* Read the whole WAV file into a single REXX variable. */
/* REXX variables are limited by available memory.      */
mydoink=doinkf~charin(1,doinkf~chars)
say 'Number of characters read into variable' mydoink~length
```

The CHARIN method returns a string of characters from the stream, which in this case is DOINK.WAV. CHARIN accepts two optional arguments. When no arguments are specified, CHARIN reads one character from the current read position and then advances the read pointer.

The first argument is a start position for reading the file. We specified 1 so that CHARIN would begin reading with the first character of the file. We could have omitted the first argument to get the same result.

The second argument tells how many characters to read. We want to read all the characters, so we specified `doinkf~chars` as the second argument. The CHARS method returns the number of characters remaining to be read in the input stream receiving the message. CHARIN then returns all the characters in the stream. DOINK.WAV has about 7500 characters.

Reading Text Files a Character at a Time

You may be wondering whether you can use the CHARIN and other character methods on text files. You can. Because you are reading the file as characters, however, CHARIN returns the line-end characters to your program. Line methods, you'll recall, don't return the line-end characters to your program.

The line-end characters on OS/2 consist of a carriage return (ASCII value of 13) and a line feed (ASCII value of 10). REXX adds these characters to the end of every line written using the LINEOUT method. Text-processing applications such as the OS/2 Enhanced Editor also add the characters. When reading a text file with CHARIN, interpret an ASCII sequence of 13 followed by 10 as the end of a line.

To see what we mean, run the following program. It writes lines to a file using LINEOUT and then reads those lines using CHARIN. (Yes, you can intermix the use of line methods and character methods.) REXX maintains separate read and write pointers, so there is no need to close the file or seek to another position before reading the lines just written.

```
/* LINECHAR.CMD -- demonstrate line end characters                */
file=.stream~new('test.dat')  /* Create a new stream object       */

file~open('both replace')  /* Open the file for reading and writing */
do i=1 to 3                /* Write three lines to the file       */
   file~lineout('Line' i)
end /* do */

do while file~chars<>0     /* Read the file a character at a time   */
   byte=file~charin        /* Read a character                      */
   ascii_value=byte~c2d    /* Convert character to a decimal value  */
   if ascii_value=13 then        /* Carriage return?               */
      say 'Carriage return'
   else if ascii_value=10 then   /* Line feed?                     */
      say 'Line feed'
   else say byte ascii_value     /* Ordinary character             */
end /* do */
file~close                    /* Close the file                    */
```

Many text-processing programs also write an end-of-file character (ASCII value 26) after the last line. You'll want to check for that character also when processing text files with character methods. To see for yourself, use the OS/2 Enhanced Editor for Presentation Manager (EPM) to edit the TEST.DAT file created by the preceding example. From EPM you don't need to make any changes. Simply save and close the file by pressing the F4 key. EPM will add the end-of-file character to the file. Then run the following program to verify it:

```
/* EPMCHAR.CMD -- demonstrate end-of-file characters              */
file=.stream~new('test.dat')  /* Create a new stream object       */

do while file~chars<>0     /* Read the file a character at a time   */
   byte=file~charin        /* Read a character                      */
   ascii_value=byte~c2d    /* Convert character to a decimal value  */
   if ascii_value=13 then        /* Carriage return?               */
      say 'Carriage return'
   else if ascii_value=10 then   /* Line feed?                     */
      say 'Line feed'
   else if ascii_value=26 then   /* End of file?                   */
      say 'End of File'
```

```
   else say byte ascii_value     /* Ordinary character              */
end /* do */
```

REXX doesn't write end-of-file characters when it closes a file that has been opened for writing.

Can you use line methods to read binary files? It's not recommended. Your binary file might not contain any new-line characters. And, if it did, the characters probably aren't meant to be interpreted as new-line characters.

Writing Binary Files

To write a binary file, use CHAROUT. CHAROUT writes only the characters that you specify in an argument to the method. CHAROUT does not add carriage-return and line-feed characters to the end of the string. Here is an example:

```
/* JACK.CMD -- demonstrate that CHAROUT does not add new-line characters */
filebin=.stream~new('binary.dat')   /* Create a new stream object       */
filebin~open('replace')             /* Open the file for replacement    */
do i=1 to 50                        /* Write fifty strings              */
  filebin~charout('All work and no play makes Jack a dull boy. ')
end
filebin~close                  /* Close the file so we can display it   */
'@echo off'                    /* Suppress echo of OS/2 commands        */
'type binary.dat'              /* Use OS/2 TYPE command to display file */
```

Because new-line characters are not added, the text displayed by the TYPE command is concatenated.

CHAROUT writes the string specified and advances the write pointer. If you want to position the write pointer before writing the string, specify the starting position as a second argument:

```
filebin~charout('Jack is loosing it.',30) /* start writing at character 30 */
```

In the example, we explicitly open and close the file. If you don't open the file, REXX attempts to open the file for both reading and writing. If you don't close the file, REXX closes it when the procedure ends. If you omit the CLOSE method in the above example, the OS/2 TYPE command won't type the file. To OS/2, the file doesn't yet exist because it is not yet closed.

Closing Files

If you don't explicitly close a file, REXX closes the file for you at the end of the procedure (that is, the end of the CMD file in which the files were opened). If your procedure is being called as an external procedure by some other REXX program, REXX will close the files before returning to the caller. In any case, it's a good idea to explicitly close files when you are done with them.

Direct File Access

REXX provides several ways for you to read records of a file directly (that is, in random order). The following example, DIRECT.CMD, shows seven different cases that illustrate some of your options.

DIRECT opens a file for both reading and writing, which is indicated by the BOTH argument on the OPEN method. The REPLACE argument on the OPEN method causes any existing DIRECT.DAT file to be replaced.

The OPEN method also has a couple of arguments we haven't yet seen: BINARY and RECLENGTH. Both of these arguments are useful for direct file access.

The BINARY argument opens the stream in binary mode, which means that line-end characters are ignored. Binary mode is useful when you want to process binary data using line methods. From studying the example you'll see that it is easier to use line methods for direct access. With line methods, you can seek to a position in a file using line numbers. With character methods, you must calculate a character displacement into the file.

The RECLENGTH argument defines a record length of 50 for the file. The RECLENGTH argument makes it possible for you to use line methods on a binary-mode stream. Since REXX now knows how long each record is, it can calculate a displacement into a file for a given record number and read the record directly.

```
/* DIRECT -- demonstration of direct file access                    */
db=.stream~new('direct.dat')
db~open('both replace binary reclength 50')

/* Write three records of 50 bytes each using LINEOUT               */
db~lineout('Cole, Gary:  Blue')
db~lineout('McGuire, Rick: Red')
db~lineout('Pritko, Steve: Red. Oops.. I mean blue!')

/* Case 1: Read the records in order using LINEIN.                  */
say 'Case 1: Sequential reads with LINEIN...'
do i=1 to 3
   say db~linein
end
say 'Press Enter to continue'; parse pull resp

/* Case 2: Read records in random order using LINEIN                */
say 'Case 2: Random reads with LINEIN...'
do i=1 to 5
   lineno=random(1,3)
   say 'Record' lineno '=' db~linein(lineno)
end
say 'Press Enter to continue'; parse pull resp

/* Case 3: Read entire file with CHARIN                             */
say 'Case 3: Read entire file with a single CHARIN...'
say db~charin(1,150)
say 'Press Enter to continue'; parse pull resp

/* Case 4: Read file sequentially with CHARIN                       */
say 'Case 4: Sequential reads with CHARIN...'
db~seek(1 read)         /* Reposition read pointer                  */
do i=1 to 3
   say db~charin(,50)
end
say 'Press Enter to continue'; parse pull resp

/* Case 5: Read records in random order with CHARIN                 */
say 'Case 5: Random reads with CHARIN...'
do i=1 to 5
   lineno=random(1,3)
   charno=((lineno-1)*50)+1
   say 'Record' lineno 'Character' charno '=' db~charin(charno,50)
end
say 'Press Enter to continue'; parse pull resp

/* Case 6: Write records in random order with LINEOUT               */
say 'Case 6: Replace record 2 with LINEOUT'
db~lineout('This should replace line 2',2)
do i=1 to 3
   say db~linein(i)
end
say 'Press Enter to continue'; parse pull resp

/* Case 7: Write records in random order with CHAROUT               */
say 'Case 7: Replace record 2 with CHARIN...'
db~charout('New record 2 from CHAROUT'~left(50,'.'),51)
db~seek(1 read)         /* Reposition read pointer                  */
do i=1 to 3
   say db~charin(,50)
end
say 'Press Enter to continue'; parse pull resp
db~close
```

After opening the file, DIRECT writes three records using LINEOUT. Notice that we didn't pad the records to 50 characters. REXX handles that for us. Because the file is opened in binary mode, REXX does not write line-end characters at the end of each line. It just writes the strings one after another to the stream.

In Case 1, the LINEIN method is used to read the file. Because the file is open in binary mode, LINEIN doesn't look for line-end characters to mark the end of a line. Instead, it relies on the record length that you specify on open. In fact, if there happened to be a carriage-return/line-feed sequence of the line, REXX would return those characters to your program.

Case 2 demonstrates how to read the file in random order. In this case, we use the RANDOM function to choose a record to retrieve. Then we specify the desired record number as an argument on LINEIN. Note that records are numbered starting from 1, not from 0. Because the file is opened in binary mode, REXX doesn't look for line-end characters. It uses the RECLENGTH to determine where to read. When used in this way, the LINEIN method can retrieve a line directly, without having to scan through the file counting line-end characters.

Case 3 proves that no line-end characters exist in the file. The CHARIN method reads the entire file. SAY displays the returned string as one long string. If REXX inserted line-end characters, each record would be displayed on a separate line.

Case 4 shows how to read the binary mode file sequentially using CHARIN. But before reading the file, we need to reset the read pointer to the beginning of the file. (Case 3 leaves the read pointer at the end of the file.) We used a SEEK method to reset the read pointer to character 1, which is the beginning of the file. As with lines, REXX numbers characters starting with 1, not 0. Position 1 is the first character of the file.

By default, the number specified on SEEK refers to a character positioning. You can also seek by line number or by offsets. SEEK allows offsets from the current read or write position, or from the beginning or ending of the file. If you prefer typing longer method names, you can use POSITION as a synonym for SEEK.

In the loop in Case 4, the first argument on CHARIN is omitted. The first argument tells where to position the read pointer. If it is omitted, REXX automatically advances the read pointer based on the number of characters you are reading.

Case 5 demonstrates how to read records in random order with CHARIN. In the loop, a random record number is selected and assigned to variable `lineno.` This record number is then converted to a character number, which can be used to specify the read position on CHARIN. Compare Case 5 with Case 2. In Case 2, which uses line methods, it isn't necessary to perform a calculation, you just request the record you want.

Cases 6 and 7 write records in random order. Case 6 uses LINEOUT, while Case 7 uses CHAROUT. Because the file is opened in binary mode, LINEOUT doesn't write line-end characters. You can write over a line by specifying a line number. With CHAROUT, you need to calculate the character position of the line to be replaced. Also unlike LINEOUT, you need to ensure that the string being written with CHAROUT is padded to the appropriate record length. Otherwise, part of the record being replaced will remain in the file.

Consequently, for random reading of files with fixed length records, line methods are often the better choice. However, one limitation of the line methods is that you cannot use them to write sparse records. That is, if a file already has 200 records, you can use LINEOUT to write record 201, but you cannot use LINEOUT to write record 300. With CHAROUT, you can open a new file and start writing at character position 5000 if you choose.

Does the File Exist?

To check for the existence of a file, use the QUERY method of the Stream class. The following ISTHERE.CMD program accepts a file name as a command line argument and checks for the existence of that file.

```
/* ISTHERE.CMD -- test for the existence of a file       */
parse arg fid                    /* Get the file name    */
qfile=.stream~new(fid)           /* Create stream object */
if qfile~query('exists')='' then /* Check for existence  */
   say fid 'does not exist.'
else
   say fid 'exists.'
```

In the example, a stream object is created for the file even though it may not exist. This is perfectly acceptable. Remember that the file is not opened when the stream object is created.

The QUERY method accepts one argument. To check for existence, specify the string `'exists'` as shown above. If the file exists, QUERY returns the full-path specification of the stream object. Otherwise, QUERY returns a null string.

Getting Other Information about a File

The QUERY method can do more than check for the existence of a file. It can return date and time stamps, read position, write position, the size of the file, and so on. The following example shows most of the QUERY arguments.

```
/* INFOON.CMD -- display information about a file */
```

```
parse arg fid
qfile=.stream~new(fid)
fullpath=qfile~query('exists')
if fullpath='' then do
   say fid 'does not exist.'
   exit
end
qfile~open('both')
say ''
say 'Full path name:' fullpath
say 'Date and time stamps (U.S. format):' qfile~query('datetime')
say '            (International format):' qfile~query('timestamp')
say ''
say 'Handle associated with stream:' qfile~query('handle')
say '                  Stream type:' qfile~query('streamtype')
say ''
say '          Size of the file (characters):' qfile~query('size')
say ' Read position (in terms of characters):' qfile~query('seek
read')
say 'Write position (in terms of characters):' qfile~query('seek
write') qfile~close
```

Using Standard I/O

All of the preceding topics dealt with reading and writing files. You can use the same methods to read from standard input (usually the keyboard) and to write to standard output (usually the display). You can also use the methods to write to the standard error stream. In Object REXX, these default streams are represented by public objects of the Monitor class: .input, .output, and .error.

The streams STDIN, STDOUT, and STDERR are transient streams. For transient streams, you cannot use any method or method argument for positioning the read and write pointers. (You can't, for example, use the SEEK method on STDOUT.)

Writing to STDOUT has the same effect as using the SAY instruction. However, the SAY instruction always writes line-end characters at the end of the string. By using the CHAROUT method to write to STDOUT, you can control when line-end characters are written.

The following example shows a modified COUNT program. COUNT was introduced earlier in this section. We've modified COUNT to display a progress indicator. For every line processed, COUNT now uses CHAROUT to display a single period. COUNT doesn't write any line-end characters, so the periods wrap to the next line when they reach the end of the line in the OS/2 window.

```
/* COUNT counts the words in a file                    */
parse arg path                    /* Get the file name */
count=0                                /* Initialize the count */
file=.stream~new(path)     /* Create a stream object for the input file */
do while file~lines<>0        /* Process each line of the file */
   text=file~linein           /* Read a line                   */
   count=count+(text~words)   /* Count blank-delimited tokens  */
   .output~charout('.')       /* Write period to STDOUT        */
end
say ''
say count
```

Reading from STDIN using LINEIN is similar to reading with the PARSE PULL instruction:

```
/* INEXAM.CMD -- example of reading STDIN with LINEIN */

/* Prompt for input with SAY and PARSE instructions   */
say 'What is your name?'
parse pull response
say 'Hi' response
say ''

/* Now prompt using LINEOUT and LINEIN                */
.output~lineout('What is your name?')
response=.input~linein
.output~lineout('Hi' response)
```

Using character methods with STDIN and STDOUT gives you more control over the reading and writing of line-end characters. In the

following example, the prompting string is written to STDOUT using CHAROUT. Because CHAROUT doesn't add any line-end characters to the stream, the display cursor is positioned after the prompt string on the same line.

```
/* INCHAR.CMD -- example of reading STDIN with CHARIN */
.output~charout('What is your name? ')
response=.input~charin(,10)
.output~charout('Hi' response)
```

CHARIN is used to read the user's response. The user's keystrokes are not returned to your program until the user presses the Enter key. In the example, a length of 10 is specified. If fewer characters than the specified length are available, CHARIN waits until they become available. Otherwise, the characters are returned to your program. As usual, CHARIN does not strip any carriage-return or line-feed characters before returning the string to your program. You can observe this with INCHAR by typing several strings that are fewer than ten characters and pressing Enter after each string:

```
[C:\]inchar
What is your name? John
Q.
Public
Hi John
Q.
Pu
```

Using Devices

You can use devices by substituting a device name (such as PRN, LPT1, LPT2, COM1, and so on) for the file name when you create a stream object. Then use line or character methods to read or write the device.

The following example sends data to a printer (device name PRN in the example). In addition to sending text data, the example also sends a control character for starting a new page. You can send other control characters or escape sequences in a similar manner. (Generally, these are listed in the manual for the device.)

Usually the control characters are characters that you cannot type at the keyboard. To use them in your program, send a D2C message to the character's ASCII value as shown in the example.

```
/* PRINTIT.CMD -- Prints a text file                                  */
say 'Type file name: '           /* prompt for a file name */
pull filename                   /* ...and get it from the user       */
infile=.stream~new(filename)
printer=.stream~new('prn:')

newpage = 12~d2c                  /* save page-eject character         */

/* Repeat this loop until no lines remain in the file */
/* and keep track of the line count with COUNT        */

do count = 1 until filename~lines = 0
   if printer~lineout(infile~linein) <>0 then do
      say 'Error: unable to write to printer'
      leave
   end
   if count // 50 = 0 then                /* if the line count is a */
      printer~charout(newpage)            /* multiple of 50, then   */
                                          /* start a new page by    */
                                          /* sending the form feed  */

end                               /* go back to the start of loop    */
                                  /* until no lines remain           */

infile~close                      /* close the file                  */
exit                              /* end the program normally        */
```

Using SOM Objects

In addition to creating your own classes with REXX, you can use classes, objects, and methods created with the OS/2 System Object Model, or SOM. You no longer need to work in C or C++ to use SOM objects.

To build SOM classes for use in REXX, you need to use the SOMobjects Developer Toolkit, Release 2.1 or later. The interface repository files must be available and specified correctly in the SOMIR environment variable. (See Building SOM classes for Use within REXX and the SOMObjects Developer Toolkit publications for more information.)

Because the OS/2 Workplace Shell is composed of SOM objects, you now also have complete control over the Workplace Shell from a REXX program.

Previously, traditional REXX provided some access to Workplace Shell objects from REXX utility functions (such as SysQueryClassList, SysCreateObject, SysSetObjectData and so on). Now you have access to all workplace classes and methods. The following example shows how to create a folder on the desktop by sending a wpclsNew message to the WPFolder class.

```
folder = .wpFolder~wpclsNew('New Folder', /* Folder will have title "New Folder" */
                            '',           /* No Specific Setup String           */
                            .wpDeskTop,   /* Create new folder on the Desktop   */
                             1)           /* Lock the object to start with      */
folder~wpOpen(0,0,0)                      /* Open the folder (Default View)     */
folder~wpClose                            /* Now close the folder.              */
```

This REXX information does not include documentation for the workplace classes and methods. They are documented with the reference information for the Workplace Shell in the OS/2 Technical Library. The documentation for the methods is in the form of C-language function calls. To use the function calls to REXX, do the following:

1. Use the function name as the REXX method name.

2. Omit the first parameter (usually a pointer to a class object or a pointer to an object) as an argument on the REXX call.

3. Use the other parameters in order. When string input is required, simply use a REXX string. You don't need to worry about supplying a zero terminator. For numeric or Boolean values, use REXX numbers (which are actually strings).

   In some cases, you'll be using several methods to do a task. Often, one of these methods will return a pointer to an object, and this pointer will be needed on other calls. To do this, assign the pointer to a regular REXX variable and then pass this variable on subsequent methods.

   For functions that require pointers to classes, use the class name as shown in the preceding example (.wpDesktop). Remember to precede the class name with a period.

In addition to Workplace Shell objects, REXX can also create objects and run methods for classes created using SOM.

Importing SOM Classes

To use a SOM class in your program, you must first import it. There are two ways to import a SOM class. You can use a ::CLASS directive, or you can send IMPORT message to the SOM class.

To import a SOM class named Animal using a directive, specify the keyword EXTERNAL as follows:

```
::class beasts external 'SOM Animal'
```

The literal string following EXTERNAL indicates where we are to import the class from and the name of that class. In this case we are importing the Animal class from SOM. The name Beasts is given to the REXX class. You can use the same name (Animal) if you wish.

The second way to import a class is by sending a message to a REXX object in the global environment named SOM. The SOM object has an IMPORT method for importing classes:

```
beastclass = .som~import('animal')
```

The argument 'animal' tells the SOM object the name of the class to import from SOM. An object representing this SOM class is returned and assigned to Beastclass. The class name in REXX will be the same as the name of the SOM class.

In addition to importing regular SOM classes, you can also import DSOM classes. Importing DSOM classes is very similar to importing a SOM class. To import using directives, you use the ::CLASS directive and must specify DSOM instead of SOM after the EXTERNAL keyword. For example, suppose the Animal class described earlier is a DSOM class. In this case you would code:

```
::CLASS beasts EXTERNAL 'DSOM Animal'
```

For dynamic loading you must first initialize the DSOM environment in REXX. You do this by using:

```
.som~SOMD_INIT
```

This installs a new object into the local environment. This object is .DSOM. It functions the same as .SOM, but for DSOM rather than SOM classes. You can now import dynamically, using:

```
beastclass = .DSOM~import('Animal')
```

Sending Messages to SOM Objects

When you import a class using REXX, the REXX class has all the methods included in the SOM class, as well as the methods in REXX's own Class class. This means you can use all the methods of the Class class *and* the SOM class. You cannot, however, create new methods for SOM classes from REXX.

For example, the NEW method (for creating new instances of a class) is part of the Class class. You can send a NEW message to Beastclass to create a new beast:

```
beast=beastclass~new
```

As part of its processing, the NEW method calls the somNew method on the SOM class to create the SOM object. NEW returns this newly created object. Use this instance of Beastclass just as you would any other REXX object. You can send it any of the SOM messages appropriate for the object. The following statement, for example, returns the name of the SOM class to which the object belongs:

```
beast~somGetClassName
```

Note that we do not need to worry about any conversions from REXX objects (Beastclass) to the real SOM object (the Animal class itself). REXX takes care of that for us.

Conversions from REXX Objects to C/SOM Method Argument Types

When a SOM method is called, a conversion needs to take place from the REXX object to a native SOM type. The method is dispatched using somApply, so some additional conversions occur because the arguments must be passed in a widened form as ANSI defines for va_lists. In general this causes no problems; however in some cases, such as with float or double parameters, a loss of precision may occur because of converting between a double (widened form) and a float. Also because of some rounding conditions, the conversion from the REXX object to a double or float may cause loss of precision.

Keep in mind that the default REXX NUMERIC DIGITS setting is 9 (See the *Object REXX Reference* for details on NUMERIC). This can cause unexpected results if you call a SOM method and pass it a value greater than 999999999 or less than -999999999. You may want to change the NUMERIC DIGITS setting, for example:

```
NUMERIC DIGITS 10
```

if you are using large long or ulong values.

The limits on the various supported SOM types follow:

TYPE	MINIMUM	MAXIMUM
boolean	0 (false)	1 (true)
char(*)	'80'x (-128)	'7f'x (127)
double	2.2250738585072014e-308	1.7976931348623158e+308
float(**)	1.175494351e-38	3.402823466e+38
long	-2147483648	2147483647
octet(*)	'00'x (0)	'ff'x (255)
short	-32768	32767
unsigned long	0	4294967295
unsigned short	0	65535

Note: The use of types char and octet is not always obvious. Because these are treated as "8-bit quantity," the parameter from REXX should be a single character. However, not all characters are exactly representable, so you may need to use the D2C and X2C built-in functions to achieve the desired results (see the *Object REXX Reference* for details about these functions).

For example, to pass the hexadecimal value 'FF'x to a SOM method as type char you would code:

```
x2c('FF'x)
```

Or, if you know the decimal value of the code, you can use it (for example, the decimal equivalent of 'FF'x is -1):

```
d2c(-1,1)
```

To convert a returned value back to the decimal equivalent, use the C2D function (see the *Object REXX Reference* for details). For type char, remember to process the value as a signed number (specify 1 as the second argument of the C2D function).

For example, if a returned value is 'FF'x, then:

```
c2d(returned_value,1)   /* Returns -2 for type char    */
c2d(returned_value)     /* Returns +254 for type octet */
```

REXX supports the following parameter or return types: void, short, ushort, long, ulong, float, double, boolean, char, string, octet, objref, pointer, struct, and somId, array, and sequence. The support for the struct type is passthru only. That is, you cannot directly change the individual elements of a structure, but you can receive it as a return and then pass it back out as a method parameter. Array and sequence parameters must be REXX array objects or be capable of returning an array with the MAKEARRAY method.

Methods Requiring INOUT and OUT Arguments

Currently REXX supports only the following types as INOUT and OUT arguments: boolean, char, octet, short, ushort, long, ulong, float, double, and string. Because INOUT and OUT arguments actually update the object data directly, you must use special objects for these arguments. REXX provides a set of SOM objects specifically for these arguments.

For each supported INOUT or OUT argument type, there is a matching class that you must use to create the object for a method requiring that type of argument. New instances of these supplied classes have a default value of 0 or '' (a null string). The following table lists the types with the matching Class names and the default values assigned to new instances:

Table 2. Class names for SOM types

CLASS NAME	CLASS TYPE	DEFAULT VALUE
boolean	DLFBoolean	0 (false)
char	DLFChar	0 (NULL)
octet	DLFOctet	0 (NULL)
short	DLFShort	0
ushort	DLFUShort	0
long	DLFLong	0
ulong	DLFULong	0
float	DLFFloat	0
double	DLFDouble	0
string	DLFString	'' (a null string)

This means that to call a method that requires an INOUT argument of type long, you must pass an object of the DLFLong class as the argument for that method. This restriction is only for INOUT and OUT arguments; for any IN argument you can specify an instance of any of the special classes or appropriate REXX classes.

To use any of the special classes within REXX you simply need to import the classes (see Importing SOM Classes).

All instances of these classes support the following methods:

_get_value
: Retrieves the current value of the object.

_set_value(newValue)
: Sets the current value of the object to *newValue* .

asString Retrieves a string representation of the current value.

In addition, DLFString has the following methods:

_get_maxSize
: Is the length of the largest string that can be stored into the object, when using this object as an INOUT or OUT parameter.

_set_maxSize(size)
: Makes *size* the length of the largest string that can be stored into the object, when using this object as an INOUT or OUT parameter. If _set_value is called with a string longer than *size* , it calls _set_maxSize with the new larger value.

Building SOM classes for Use within REXX

To build SOM classes for use in REXX, you need to use the SOMobjects Developer Toolkit, Release 2.1 or later, or the Warp Toolkit. You must build the classes into a DLL that can be dynamically loaded, and the loading of the DLL should result in the building of all SOM classes within that DLL. The *SOMobjects Users Guide* explains how to do this. It is necessary because REXX calls the somFindClass on the SOMClassMgrObject to locate the SOM classes on import. The dllname modifier must be in your IDL file so that SOMClassMgrObject knows which DLL to load for the class.

You also need to define the class in the Interface Repository, so that REXX can get information about the class's methods, method parameters, and return types.

REXX and the Workplace Shell

If you are an OS/2 user, you are familiar with the Workplace Shell. The shell provides many objects, including folders, programs, and special-purpose objects such as the shredder. You can manipulate objects with actions such as double-click and dragging and dropping one object on another. These actions send messages to the shell objects. For example, double-click sends a "wpopen" message. Objects respond to these messages by running methods that perform the requested actions. The class of the object receiving the message determines the method used for each action; therefore, opening a folder does something different than opening a program.

Sending Messages

You can use REXX to program the shell objects by sending them messages. This lets programs control actions that direct user interaction generally controls. For example, you can create a folder and place a number of program objects in it. REXX supports both procedural and object-oriented access to the Workplace Shell objects.

Procedural Support

The RexxUtil package includes several functions useful for manipulating the shell. The RexxUtil functions are very easy to use, but are not as extensive as the object-oriented support.

Object-Oriented Support

Because the Workplace Shell objects are implemented using SOM, REXX provides very powerful and flexible direct access to the shell objects, but accessing the shell objects directly requires programmers to have considerably more knowledge of how the Workplace Shell works. In addition, programming Workplace Shell objects always involves some risk of damaging the shell in a way that might require re-installation of OS/2.

As with other SOM classes, the REXX documentation does not include documentation for the workplace classes and methods. These are documented in the OS/2 Technical Library, and the documentation for the methods is in the form of C-language function calls. Before trying any object-oriented REXX programming of the Workplace Shell, you should read this reference material. REXX provides interfaces to the objects, but cannot enforce correct Workplace Shell programming.

Do not use any Workplace Shell methods containing INOUT or OUT parameters (for example, wpQueryProgDetails).

Some Workplace Shell objects by default cannot be deleted. You may want to issue a wpSetup method to the new object and to make it able to be deleted:

```
wpShredder = .wps~import('WPShredder')
xshred = wpShredder~new('myShred','',folder,1)
xshred~wpsetup('NODELETE=NO')
```

REXX currently does not support complex Workplace Shell parameters; it supports only string, integer, or boolean. (For example, it cannot support PPROGDETAILS.)

REXX Workplace Shell initialization adds a hidden object, OrxInit, to your desktop. This is a WPZORXINIT object. You will notice this only if you query the contents of the desktop folder.

Accessing Workplace Shell Objects

REXX automatically imports several workplace classes and adds them to the global environment. These include:

WPObject
WPTransient
WPAbstract
WPProgram
WPFolder

You can import other workplace classes just as you can import SOM classes, except for Workplace Shell classes the class server is WPS instead of SOM. Here are two ways to import the WPClock class:

```
::class WPClock external 'WPS WPClock'

WPClock = .wps~import('WPClock')
```

REXX also creates an entry in the environment for one other workplace object, WPDesktop. The other explicitly named system folders can be added to the environment by running the WPSYSOBJ.CMD utility. The REXX folder names followed by their workplace functions are:

WPNoWhere
: The hidden folder

WPOS2SYS
: The system folder

WPTemps
: The templates folder

WPStart The startup folder

WPInfo The information folder

WPDrives
: The drives folder

WPConfig
: The system setup folder

After calling WPSYSOBJ.CMD, you can refer to the information folder directly by using .wpinfo.

It is easy to locate other items on the desktop using the WPFIND.CMD utility. You can use WPFIND directly from a command line or within a program. From a command line, the syntax is:

```
wpfind -p name path
```

The *path* is the title of a workplace object to locate and place into the global environment *name* specifies.

From within a program, the syntax is:

```
obj = wpfind(path)
```

The *path* is the title of a workplace object that *obj* will reference.

The path must always be a string. If you are looking for a folder with the title "Folder2" and know it is inside the folder entitled "Folder1," which resides on the desktop, you can call it as follows:

```
is_found=wpfind('Folder1;Folder2')
```

Another way to call WPFIND is to use the folder itself directly. If *f1* is the folder entitled "Folder1" and *f2* is the folder entitled "Folder2," the call:

```
is_found = wpfind(f1~wpQueryTitle';'f2~wpQueryTitle)
```

changes everything in parenthesis to a valid string.

The basic assumption is that the very first folder that using WPFIND searches is .wpDesktop. WPFIND returns the object itself. If the requested object is not found, WPFIND returns the NIL object.

The Workplace documentation frequently uses named constant values. REXX provides the WPCONST.CMD utility to let programmers use the named constants instead of looking up their values. Running WPCONST.CMD sets up a directory called WPCONST in the global environment that contains the names and their values. For example, the OPEN_SETTINGS constant could be referenced as:

```
.wpconst['OPEN_SETTINGS'].
```

Note: Use quotation marks around constant names, such as "OPEN_SETTINGS" in the preceding example.

Workplace Shell Example: Creating Folders

Here is a simple example showing how to create a folder and add objects to it:

```
/* Add the workplace constants to the global environment         */
call wpconst

/* Create a new folder with title 'My Folder' on the desktop     */
newfolder = .wpfolder~new('My Folder','',.wpdesktop,1)
```

You may have to rearrange your windows a little bit to make the new folder's icon visible.

```
/* Send a message to the new folder object and show the result   */
say newfolder~wpquerytitle

/* Change the title of the new folder                            */
newfolder~wpsettitle('My Excellent Folder')

/* Add a program object to the new folder                        */
.wpprogram~new('REXXTRY','EXENAME=pmrexx.exe;PARAMETERS=REXXTRY',newfolder,1)

/* Open the new folder and make it current                       */
newfolder~~wpopen(0,.wpconst['OPEN_DEFAULT'],0)~wpswitchto(0)
```

Now double click the REXXTRY icon to verify that the program was added correctly and works as expected.

```
/* All done, close the folder                                    */
newfolder~wpclose
```

A Workplace Shell Example: Animated Icons

Here is another example showing how to animate a folder's icon.

```
/* Put the workplace system folder names into the environment    */
```

```
call wpsysobj

sysicon = .wpos2sys~wpqueryicon

configicon = .wpconfig~wpqueryicon

/* Create a new folder with title 'My Folder' on the desktop     */
newfolder = .wpfolder~new('My Folder','',.wpdesktop,1)

do 20;newfolder~wpseticon(sysicon);newfolder~wpseticon(configicon);end
```

For additional examples, see the SOM section of the *Object REXX Reference.*

REXX Application Programming Interfaces

This appendix describes how to interface applications to REXX or extend the REXX language by using REXX application programming interfaces (APIs). As used here, the term *application* refers to programs written in languages other than REXX. Commonly this is the C language. Conventions in this appendix are based on the C language. Refer to a C programming reference manual if you need a better understanding of these conventions.

The features described here let an application extend many parts of the REXX language or extend an application with REXX. This includes creating handlers for subcommands, external functions, and system exits.

Subcommands
: are commands issued from a REXX program. A REXX expression is evaluated and the result is passed as a command to the currently "addressed" subcommand handler. Subcommands are used in REXX programs running as application macros.

Functions
: are direct extensions of the REXX language. An application can create functions that extend the native REXX function set. Functions may be general- purpose extensions or specific to an application.

System exits
: are programmer-defined variations of the operating system. The application programmer can tailor the REXX interpreter behavior by replacing REXX system requests.

Subcommand, function, and system exit handlers have similar coding, compilation, and packaging characteristics.

In addition, applications can manipulate the variables in REXX programs (see Variable Pool Interface), and execute REXX routines directly from memory (see Macrospace Interface).

Handler Characteristics

The basic requirements for subcommand, function, and system exit handlers are:

- REXX handlers must use the system linkage convention. Handler functions should be declared with the appropriate type definition from the REXXSAA.H include file:
  - RexxSubcomHandler
  - RexxFunctionHandler
  - RexxExitHandler
- A REXX handler must be packaged as either:
  - An exported routine within a Dynamic Link Library (DLL)
  - An entry point within an executable (EXE) module
- A handler must be registered with REXX before it can be used. REXX uses the registration information to locate and call the handler. For example, external function registration of a dynamic link library external function identifies both the dynamic link library and routine that contains the external function. Also note:

- Dynamic link library handlers are global to the OS/2 system; any REXX program can call them.
- EXE file handlers are local to the registering process; only a REXX program running in the same process as an EXE module can call a handler packaged within that EXE module.

RXSTRINGs

Many of the REXX application programming interfaces pass REXX character strings to and from a REXX procedure. The RXSTRING data structure is used to describe REXX character strings. An RXSTRING is a content-insensitive, flat model character string with a theoretical maximum length of 4 gigabytes. The following structure defines an RXSTRING:

RXSTRING data structure

```
typedef struct {
   ULONG           strlength;     /* length of string            */
   PCH             strptr;        /* pointer to string           */
} RXSTRING;
typedef RXSTRING *PRXSTRING;      /* pointer to an RXSTRING      */
```

Notes:

1. The REXXSAA.H include file contains a number of convenient macros for setting and testing RXSTRING values.
2. An RXSTRING can have a value (including the null string, "") or it can be empty.
   - If an RXSTRING has a value, the *strptr* field is non-NULL. The RXSTRING macro RXVALIDSTRING(string) returns TRUE.
   - If an RXSTRING is the REXX null string (""), the *strptr* field is non-NULL and the *strlength* field is zero. The RXSTRING macro RXZEROLENSTRING(string) returns TRUE.
   - If an RXSTRING is empty, the field *strptr* is NULL. The RXSTRING macro RXNULLSTRING(string) returns TRUE.
3. When the REXX interpreter passes an RXSTRING to a subcommand handler, external function, or exit handler, the interpreter adds a null character (hexadecimal zero) at the end of the RXSTRING data. You can use the C string library functions on these strings. However, the RXSTRING data may also contain null characters. There is no guarantee that the first null character encountered in an RXSTRING marks the end of the string. Use the C string functions only when you do not expect null characters in the RXSTRINGs (such as file names passed to external functions). The *strlength* field in the RXSTRING does not include the terminating null character.
4. On calls to subcommand and external functions handlers, as well as some of the exit handlers, the REXX interpreter expects an RXSTRING value returned. The REXX interpreter provides a default RXSTRING with a *strlength* of 256 for the returned information. If the returned data is shorter than 256 characters, the handler can copy the data into the default RXSTRING and set the *strlength* field to the length returned.

   If the returned data is longer than 256 characters, a new RXSTRING can be allocated using DosAllocMem. The *strptr* field must point to the new storage and the *strlength* must be set to the string length. The REXX interpreter will return the newly allocated storage to the system for the handler routine.

Calling the REXX Interpreter

A REXX program may be run directly by the operating system or from within an application program.

From the Operating System

The CMD.EXE command shell calls the REXX interpreter for the user:

- At command prompts
- In calls from CMD (batch) files

  Note: Use the OS/2 CALL command to call a REXX program in a batch file if you want control to return to the caller.
- From the object that represents the program

From within an Application

The REXX interpreter is an dynamic link library (DLL) routine. Any application can call the REXX interpreter to run a REXX program. The interpreter is fully re-entrant and supports REXX procedures running on multiple threads within the same process.

A C-language prototype for calling REXX is in the REXXSAA.H include file.

The RexxStart Function

RexxStart calls the REXX interpreter to run a REXX procedure.

Topics:

RexxStart - Syntax

**RexxStart(ArgCount, ArgList, ProgramName, Instore, EnvName, CallType,
Exits, ReturnCode, Result)**

RexxStart - Parameters

ArgCount *(LONG) - input*

The number of elements in the *ArgList* array. This is the value that the *ARG()* built-in function in the REXX program returns. *ArgCount* includes RXSTRINGs that represent omitted arguments. Omitted arguments are empty RXSTRINGs (*strptr* is NULL).

ArgList *(PRXSTRING) - input*

An array of RXSTRING structures that are the REXX program arguments.

ProgramName *(PSZ) - input*

Address of the ASCIIZ name of the REXX procedure. If *Instore* is NULL, *ProgramName* must contain at least the file name of the REXX procedure. You can also provide an extension, drive, and path specification. If you do not specify a file extension, the default is ".CMD". A REXX program can use any extension. If you do not provide the path and drive, the REXX interpreter uses the usual file search (current directory, then environment path).

If *Instore* is not NULL, *ProgramName* is the name used in the PARSE SOURCE instruction. If *Instore* requests a REXX procedure from the macrospace, *ProgramName* is the macrospace function name (see Macrospace Interface).

Instore *(PRXSTRING) - input*

An array of two RXSTRING descriptors for in-storage REXX procedures. If the *strptr* fields of both RXSTRINGs are NULL, the interpreter searches for REXX procedure *ProgramName* in the REXX macrospace (see Macrospace Interface). If the procedure is not in the macrospace, the call to RexxStart terminates with an error return code.

If either *Instore strptr* field is not NULL, *Instore* is used to run a REXX procedure directly from storage.

Instore[0]

An RXSTRING describing a memory buffer containing the REXX procedure source. The source must be an exact image of a REXX procedure disk file (complete with carriage returns, line feeds, and end-of-file characters).

Instore[1]

An RXSTRING containing the translated image of the REXX procedure. If *Instore[1]* is empty, the REXX interpreter returns the translated image in *Instore[1]* when the REXX procedure finishes running. The translated image may be used in *Instore[1]* on subsequent RexxStart calls.

If *Instore[1]* is not empty, the interpreter runs the translated image directly. The program source provided in *Instore[0]* is used only if the REXX procedure uses the SOURCELINE built-in function. *Instore[0]* can be empty if SOURCELINE is not used. If *Instore[0]* is empty and the procedure uses the SOURCELINE built-in function, SOURCELINE() returns 0 lines and any attempt to access the source returns Error 40.

If *Instore[1]* is not empty, but does not contain a valid REXX translated image, unpredictable results can occur. The REXX interpreter may be able to determine that the translated image is incorrect and retranslate the source.

Instore[1] is both an input and an output parameter.

If the procedure is executed from disk, the *Instore pointer* must be NULL. If the first argument string in *Arglist* contains the string "//T" and the *CallType* is RXCOMMAND, the interpreter translates the procedure source and writes the translated image to disk. Since the *Instore pointer* is NULL, the translated image is not returned in an *Instore* parameter.

The program calling RexxStart must release *Instore[1]* using DosFreeMem when the translated image is no longer needed.

The format of the translated image of a REXX program is not a programming interface. Only the interpreter version that created the image can run the translated image. Therefore, do not move a translated image to other systems or save it for later use. You can, however, use the translated image multiple times during a single application instance.

EnvName *(PSZ) - input*

Address of the ASCIIZ initial ADDRESS environment name. The ADDRESS environment is a subcommand handler registered using RexxRegisterSubcomExe or RexxRegisterSubcomDll. *EnvName* is used as the initial setting for the REXX ADDRESS instruction.

If *EnvName* is NULL, the file extension is used as the initial ADDRESS environment. The environment name cannot be longer than 250 characters.

CallType *(LONG) - input*

The type of REXX procedure execution. Allowed execution types are:

RXCOMMAND The REXX procedure is a system or application command. REXX commands usually have a single argument string. The REXX PARSE SOURCE instruction returns COMMAND as the second token.

`RXSUBROUTINE` The REXX procedure is a subroutine of another program. The subroutine can have multiple arguments and does not need to return a result. The REXX PARSE SOURCE instruction returns `SUBROUTINE` as the second token.

`RXFUNCTION` The REXX procedure is a function called from another program. The subroutine may have multiple arguments and must return a result. The REXX PARSE SOURCE instruction returns `FUNCTION` as the second token.

Exits *(PRXSYSEXIT) - input*
An array of RXSYSEXIT structures defining exits for the REXX interpreter to use. The RXSYSEXIT structures have the following form:

```
typedef struct {
   PSZ            sysexit_name;  /* name of exit handler        */
   LONG           sysexit_code;  /* system exit function code   */
} RXSYSEXIT;
```

The *sysexit_name* is the address of an ASCIIZ exit handler name registered with RexxRegisterExitExe or RexxRegisterExitDll. *Sysexit_code* is a code identifying the handler exit type. System Exits provides exit code definitions. An RXENDLST entry identifies the system exit list end. *Exits* must be NULL if exits are not used.

ReturnCode *(PSHORT) - output*
The integer form of the *Result* string. If the *Result* string is a whole number in the range -(2**15) to 2**15-1, it is converted to an integer and also returned in *ReturnCode* .

Result *(PRXSTRING) - output*
The string returned from the REXX procedure with the REXX RETURN or EXIT instruction. A default RXSTRING can be provided for the returned result. If a default RXSTRING is not provided or the default is too small for the returned result, the REXX interpreter allocates an RXSTRING using DosAllocMem. The caller of RexxStart is responsible for releasing the RXSTRING storage with DosFreeMem.

The REXX interpreter does not add a terminating null to *Result* .

RexxStart - Returns

The RexxStart return values are:

negative	Interpreter errors. See Appendix A in the *Object REXX Reference* for the list of REXX errors.
0	No errors occurred. The REXX procedure ran normally.
positive	A system return code indicating problems finding or loading the interpreter. See the return codes for the OS/2 functions DosLoadModule and DosQueryProcAddr for details.

When a macrospace REXX procedure (see Macrospace Interface) is not loaded in the macrospace, the return code is -3 ("Program is unreadable").

RexxStart - Example

The following is an example of a sample call to the REXX interpreter:

Sample call to the REXX interpreter

```
LONG      return_code;               /* interpreter return code    */
RXSTRING  argv[1];                   /* program argument string    */
```

```
RXSTRING  retstr;                     /* program return value       */
SHORT     rc;                         /* converted return code      */
CHAR      return_buffer[250];         /* returned buffer            */
                                      /* build the argument string  */
  MAKERXSTRING(argv[0], macro_argument,
      strlen(macro_argument));
                                      /* set up default return      */
  MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));
  return_code = RexxStart(1,          /* one argument               */
                          argv,       /* argument array             */
                          "CHANGE.ED", /* REXX procedure name       */
                          NULL,       /* use disk version           */
                          "Editor",   /* default address name       */
                          RXCOMMAND,  /* calling as a subcommand    */
                          NULL,       /* no exits used              */
                          &rc,        /* converted return code      */
                          &retstr);   /* returned result            */
                                      /* process return value       */
.
.
.
                                      /* need to return storage?    */
  if (RXSTRPTR(retval) != return_buffer)
    DosFreeMem(RXSTRPTR(retval));     /* release the RXSTRING       */
```

Subcommand Interface

An application can create handlers to process commands from a REXX program. Once created, the subcommand handler name can be used with the RexxStart function or the REXX ADDRESS instruction. Subcommand handlers must be registered with the RexxRegisterSubcomExe or RexxRegisterSubcomDll function before use.

Registering Subcommand Handlers

A subcommand handler can reside in the same module (EXE or DLL) as an application, or it can reside in a separate dynamic link library. An application that runs REXX procedures with RexxStart should use RexxRegisterSubcomExe to register subcommand handlers. The REXX interpreter passes commands to the application subcommand handler entry point. Subcommand handlers created using RexxRegisterSubcomExe are available only to REXX programs called from the registering application.

The RexxRegisterSubcomDll interface creates subcommand handlers that reside in a dynamic link library. Any REXX program using the REXX ADDRESS instruction can access a dynamic link library subcommand handler. A dynamic link library subcommand handler can also be registered directly from a REXX program using the RXSUBCOM command.

Creating Subcommand Handlers

The following example is a sample subcommand handler definition.

Sample definition of a subcommand handler

```
ULONG command_handler(
  PRXSTRING Command,    /* Command string from REXX           */
  PUSHORT   Flags,      /* Returned Error/Failure flags       */
  PRXSTRING Retstr);    /* Returned RC string                 */
```

Command The command string created by REXX.

Command is a null-terminated RXSTRING containing the issued command.

Flags — Subcommand completion status. The subcommand handler can indicate success, error, or failure status. The subcommand handler can set *Flags* to one of the following values:

RXSUBCOM_OK

The subcommand completed normally. No errors occurred during subcommand processing and the REXX procedure continues when the subcommand handler returns.

RXSUBCOM_ERROR

A subcommand error occurred. RXSUBCOM_ERROR indicates a subcommand error occurred; for example, incorrect command options or syntax.

If the subcommand handler sets *Flags* to RXSUBCOM_ERROR, the REXX interpreter raises an ERROR condition if SIGNAL ON ERROR or CALL ON ERROR traps have been created. If TRACE ERRORS has been issued, REXX traces the command when the subcommand handler returns.

RXSUBCOM_FAILURE

A subcommand failure occurred. RXSUBCOM_FAILURE indicates that general subcommand processing errors have occurred. For example, unknown commands usually return RXSUBCOM_FAILURE.

If the subcommand handler sets *Flags* to RXSUBCOM_FAILURE, the REXX interpreter raises a FAILURE condition if SIGNAL ON FAILURE or CALL ON FAILURE traps have been created. If TRACE FAILURES has been issued, REXX traces the command when the subcommand handler returns.

Retstr — Address of an RXSTRING for the return code. *Retstr* is a character string return code that is assigned to the REXX special variable RC when the subcommand handler returns to REXX. The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING may be allocated with DosAllocMem if the return string is longer than the default RXSTRING. If the subcommand handler sets *Retstr* to an empty RXSTRING (a NULL *strptr*), REXX assigns the string "0" to RC.

Example

The following is a sample subcommand handler:

Sample subcommand handler

```
ULONG Edit_Commands(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer to short for return of flags     */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  LONG      command_id;               /* command to process         */
  LONG      rc;                       /* return code                */
  PSZ       scan_pointer;             /* current command scan       */
  PSZ       target;                   /* general editor target      */
  scan_pointer = command->strptr;     /* point to the command       */
                                      /* resolve command            */
  command_id = resolve_command(&scan_pointer);
  switch (command_id) {               /* process based on command   */
    case   LOCATE:                    /* locate command             */
                                      /* validate rest of command   */
      if (rc = get_target(&scan_pointer, &target)) {
        *Flags = RXSUBCOM_ERROR;      /* raise an error condition   */
        break;                        /* return to REXX             */
      }
      rc = locate(target);            /* look target in the file    */
      *Flags = RXSUBCOM_OK;           /* not found is not an error  */
      break;                          /* go finish up               */
.
.
```

.

```
    default:                          /* unknown command            */
      rc = 1;                         /* return code for unknown    */
      *Flags = RXSUBCOM_FAILURE;      /* this is a command failure  */
      break;
  }
  sprintf(Retstr->strptr, "%d", rc);  /* format return code string  */
                                      /* and set the correct length */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                           /* processing completed       */
}
```

Subcommand Interface Functions

The functions for registering and using subcommand handlers are:

RexxRegisterSubcomDll
RexxRegisterSubcomExe
RexxDeregisterSubcom
RexxQuerySubcom

RexxRegisterSubcomDll

RexxRegisterSubcomDll registers a subcommand handler that resides in a dynamic link library routine.

Topics:

- Syntax
- Parameters
- Returns
- Remarks

RexxRegisterSubcomDll - Syntax

RexxRegisterSubcomDll(EnvName, ModuleName, EntryPoint, UserArea, DropAuth)

RexxRegisterSubcomDll - Parameters

EnvName *(PSZ) - input*

Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the subcommand handler routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynamic link library procedure name. *EntryPoint* is the name of the exported routine within *ModuleName* that REXX calls as a subcommand handler.

UserArea *(PUCHAR) - input*
Address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the subcommand handler registration. *UserArea* can be NULL if there is no user information to save. The RexxQuerySubcom function can retrieve the saved user information.

DropAuth *(ULONG) - input*
The drop authority. *DropAuth* identifies the processes that can deregister the subcommand handler. The possible *DropAuth* values are:

RXSUBCOM_DROPPABLE
Any process can deregister the subcommand handler with RexxDeregisterSubcom.

RXSUBCOM_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterSubcom.

RexxRegisterSubcomDll - Returns

The RexxRegisterSubcomDll return values are:

0	RXSUBCOM_OK	A subcommand has executed successfully.
10	RXSUBCOM_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this subcommand you must specify its library name.)
1002	RXSUBCOM_NOEMEM	There is insufficient memory to complete this request.

RexxRegisterSubcomDll - Remarks

EntryPoint can be either a 16-bit or a 32-bit routine. REXX calls the handler in the correct addressing mode.

RexxRegisterSubcomExe

RexxRegisterSubcomExe registers a subcommand handler that resides within application code.

Topics:

RexxRegisterSubcomExe - Syntax

RexxRegisterSubcomExe(EnvName, EntryPoint, UserArea)

RexxRegisterSubcomExe - Parameters

EnvName *(PSZ) - input*
: Address of an ASCIIZ subcommand handler name.

EntryPoint *(PFN) - input*
: Address of the subcommand handler entry point within the application EXE code.

UserArea *(PUCHAR) - input*
: Address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the subcommand handler registration. *UserArea* can be NULL if there is no user information to save. The RexxQuerySubcom function can retrieve the user information.

RexxRegisterSubcomExe - Returns

The RexxRegisterSubcomExe return values are:

0	RXSUBCOM_OK	A subcommand has executed successfully.
10	RXSUBCOM_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this subcommand you must specify its library name.)

30	RXSUBCOM_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other REXX subcommand functions).
1002	RXSUBCOM_NOEMEM	There is insufficient memory to complete this request.

RexxRegisterSubcomExe - Remarks

If *EnvName* is the same as a subcommand handler already registered with RexxRegisterSubcomDll, RexxRegisterSubcomExe returns RXSUBCOM_DUP. This is not an error condition. RexxRegisterSubcomExe has successfully registered the new subcommand handler.

A REXX procedure can register dynamic link library subcommand handlers with the RXSUBCOM command. For example:

```
                               /* register Dialog Manager       */
                               /* subcommand handler            */
  'RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR'
  Address ispcir               /* send commands to dialog mgr   */
```

The RXSUBCOM command registers the Dialog Manager subcommand handler ISPCIR as routine ISPCIR in the ISPCIR dynamic link library.

RexxRegisterSubcomExe - Example

The following is a sample subcommand handler registration:

Sample subcommand handler registration

```
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  user_info[0] = global_workarea;      /* save global work area for  */
  user_info[1] = NULL;                 /* re-entrancy                */
  rc = RexxRegisterSubcomExe("Editor", /* register editor handler    */
      &Edit_Commands,                  /* located at this address    */
      user_info);                      /* save global pointer        */
```

RexxDeregisterSubcom

RexxDeregisterSubcom deregisters a subcommand handler.

Topics:

- Syntax
- Parameters

RexxDeregisterSubcom - Syntax

RexxDeregisterSubcom(EnvName, ModuleName)

RexxDeregisterSubcom - Parameters

EnvName *(PSZ) - input*
: Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
: Address of an ASCIIZ dynalink library name. *ModuleName* is the name of the dynalink library containing the registered subcommand handler. When *ModuleName* is NULL, RexxDeregisterSubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxDeregisterSubcom does not find a RexxRegisterSubcomExe handler, it searches the RexxRegisterSubcomDll subcommand handler list.

RexxDeregisterSubcom - Returns

The RexxDeregisterSubcom return values are:

0	RXSUBCOM_OK	A subcommand has executed successfully.
30	RXSUBCOM_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other REXX subcommand functions).
40	RXSUBCOM_NOCANDROP	The subcommand handler has been registered as "not droppable."

RexxDeregisterSubcom - Remarks

The handler is removed from the active subcommand handler list.

RexxQuerySubcom

RexxQuerySubcom queries a subcommand handler and retrieves saved user information.

Topics:

RexxQuerySubcom - Syntax

RexxQuerySubcom(EnvName, ModuleName, Flag, UserWord)

RexxQuerySubcom - Parameters

EnvName *(PSZ) - input*
: Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
: Address of an ASCIIZ dynamic link library name. *ModuleName* restricts the query to a subcommand handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxQuerySubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxQuerySubcom does not find a RexxRegisterSubcomExe handler, it searches the RexxRegisterSubcomDll subcommand handler list.

Flag *(PUSHORT) - output*
: Subcommand handler registration flag. *Flag* is the *EnvName* subcommand handler registration status. When RexxQuerySubcom returns RXSUBCOM_OK, the *EnvName* subcommand handler is currently registered. When RexxQuerySubcom returns RXSUBCOM_NOTREG, the *EnvName* subcommand handler is not registered.

UserWord *(PUCHAR) - output*
: Address of an 8-byte area to receive the user information saved with RexxRegisterSubcomExe or RexxRegisterSubcomDll. *UserWord* can be NULL if the saved user information is not required.

RexxQuerySubcom - Returns

The RexxQuerySubcom return values are:

0	RXSUBCOM_OK	A subcommand has executed successfully.
30	RXSUBCOM_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other REXX subcommand functions).

RexxQuerySubcom - Example

The following is a sample subcommand handler query:

Sample subcommand handler query

```
ULONG Edit_Commands(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer to short for return of flags     */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */
  rc = RexxQuerySubcom("Editor",       /* retrieve application work  */
      NULL,                            /* area anchor from REXX.     */
      &query_flag,
      user_info);
  global_workarea = user_info[0];      /* set the global anchor      */
```

Subcommand Interface Returns

Return codes for the Subcommand Interface are as follows:

0x01	RXSUBCOM_ERROR	An error in subcommand processing has occurred; the interpreter raises an ERROR condition.
0x02	RXSUBCOM_FAILURE	A failure in subcommand processing has occurred; the interpreter raises a FAILURE condition.
0	RXSUBCOM_OK	A subcommand has executed successfully.
10	REXSUBCOM_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this subcommand you must specify its library name.)
30	RXSUBCOM_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names

		(RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other REXX subcommand functions).
40	RXSUBCOM_NOCANDROP	The subcommand handler has been registered as "not droppable."
50	RXSUBCOM_LOADERR	An error has occurred while loading a dynalink library; most commonly, a missing dynalink library file causes the error.
1002	RXSUBCOM_NOEMEM	There is insufficient memory to complete this request.

External Function Interface

There are two types of REXX external functions:

1. Routines written in REXX
2. Routines written in other OS/2-supported languages.

External functions written in REXX do not need to be registered. These functions are found by a disk search for a REXX procedure file that matches the function name.

Registering External Functions

An external function can reside in the same module (EXE or DLL) as an application, or in a separate dynamic link library. RexxRegisterFunctionExe registers external functions within an application module. External functions registered with RexxRegisterFunctionExe are available only to REXX programs called from the registering application.

The RexxRegisterFunctionDll interface registers external functions that reside in a dynamic link library. Any REXX program can access a dynamic link library external function after it is registered. A dynamic link library external function can also be registered directly from a REXX program using the REXX RXFUNCADD built-in function.

Creating External Functions

The following is a sample external function definition:

Sample external function definition

```
ULONG SysLoadFuncs(
     PSZ       Name,                 /* name of the function       */
     LONG      Argc,                 /* number of arguments        */
     RXSTRING  Argv[],               /* list of argument strings   */
     PSZ       Queuename,            /* current queue name         */
     PRXSTRING Retstr)               /* returned result string     */
```

Name	Address of ASCIIZ function name used to call the external function.
Argc	The number of elements in the *Argv* array. *Argv* will contain *Argc* RXSTRINGs.

Argv	An array of null-terminated RXSTRINGs for the function arguments.
Queuename	The name of the currently defined REXX external data queue.
Retstr	Address of an RXSTRING for the returned value. *Retstr* is a character string function or subroutine return value. When a REXX program calls an external function with the REXX CALL instruction, *Retstr* is assigned to the REXX special variable RESULT. When the REXX program calls an external function with a function call, *Retstr* is used directly within the REXX expression. The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING can be allocated with DosAllocMem if the returned string is longer than 256 bytes. The REXX interpreter releases *Retstr* with DosFreeMem when the external function completes.
Returns	An integer return code from the function. When the external function returns `0`, the function completed successfully. *Retstr* contains the function return value. When the external function returns a non-zero return code, the REXX interpreter raises REXX error 40 ("Incorrect call to routine"). The *Retstr* value is ignored. If the external function does not have a return value, the function should set *Retstr* to an empty RXSTRING (NULL *strptr*). When an external function called as a function does not return a value, the interpreter raises error 44, "Function or message did not return data". When an external function called with the REXX CALL instruction does not return a value, the REXX interpreter drops (unassigns) the special variable RESULT.

Calling External Functions

RexxRegisterFunctionExe external functions are local to the registering process. Only REXX procedures running in the same process can call the registered external function. It is possible to register functions with the same external function name if they are registered from different processes. However, RexxRegisterFunctionDll functions are available from all processes. The function names cannot be duplicated.

Example

The following is a sample external function routine:

Sample external function routine

```
ULONG SysMkDir(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
{
  ULONG  rc;                          /* Return code of function    */
  if (Argc != 1)                      /* must be 1 argument         */
    return 40;                        /* incorrect call if not      */
                                      /* make the directory using   */
                                      /* the null-terminated        */
  rc = DosMkDir(Argv[0].strptr, 0L);  /* directly                   */
  sprintf(Retstr->strptr, "%d", rc);  /* result is return code      */
                                      /* set proper string length   */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                           /* successful completion      */
}
```

External Functions Interface Functions

The functions for registering and using external functions are:

RexxRegisterFunctionDll
RexxRegisterFunctionExe
RexxDeregisterFunction
RexxQueryFunction

RexxRegisterFunctionDll

RexxRegisterFunctionDll registers an external function that resides in a dynamic link library routine.

Topics:

- Syntax
- Parameters
- Returns
- Remarks
- Example

RexxRegisterFunctionDll - Syntax

RexxRegisterFunctionDll(FuncName, ModuleName, EntryPoint)

RexxRegisterFunctionDll - Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the external function routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynamic link procedure name. *EntryPoint* is the name of the exported external function routine within *ModuleName* .

RexxRegisterFunctionDll - Returns

The RexxRegisterFunctionDll return values are:

0	RXFUNC_OK	The call to the function completed successfully.
10	RXFUNC_DEFINED	The requested function is already registered.
20	RXFUNC_NOMEM	There is not enough memory to register a new function.

RexxRegisterFunctionDll - Remarks

EntryPoint can be either a 16-bit or 32-bit routine. REXX calls the function in the correct addressing mode.

A REXX procedure can register dynamic link library subcommand handlers with the RXFUNCADD built-in function. For example:

```
                                /* register function SysLoadFuncs*/
                                /* in dynalink library REXXUTIL  */
  Call RxFuncAdd 'SysLoadFuncs', 'REXXUTIL', 'SysLoadFuncs'
  Call SysLoadFuncs             /* call to load other functions  */
```

RXFUNCADD registers the external function SysLoadFuncs as routine SysLoadFuncs in the REXXUTIL dynamic link library. SysLoadFuncs registers additional functions in REXXUTIL.DLL with RexxRegisterFunctionDll. See the SysLoadFuncs routine below for a function registration example.

RexxRegisterFunctionDll - Example

The following is a sample function package load routine:

Function package load routine

```
static PSZ  RxFncTable[] =            /* function package list      */
{
      "SysCls",
      "SysCurpos",
      "SysCurState",
      "SysDriveInfo",
}
ULONG SysLoadFuncs(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
{
  INT    entries;                     /* Num of entries             */
  INT    j;                           /* Counter                    */
  Retstr->strlength = 0;              /* set null string return     */
  if (Argc > 0)                       /* check arguments            */
    return 40;                        /* too many, raise an error   */
```

```
                                     /* get count of arguments     */
  entries = sizeof(RxFncTable)/sizeof(PSZ);
                                     /* register each function in  */
  for (j = 0; j < entries; j++) {    /* the table                  */
    RexxRegisterFunctionDll(RxFncTable[j],
          "REXXUTIL", RxFncTable[j]);
  }
  return 0;                          /* successful completion      */
}
```

RexxRegisterFunctionExe

RexxRegisterFunctionExe registers an external function that resides within the application code.

Topics:

RexxRegisterFunctionExe - Syntax

RexxRegisterFunctionExe(FuncName, EntryPoint)

RexxRegisterFunctionExe - Parameters

FuncName *(PSZ) - input*
: Address of an ASCIIZ external function name.

EntryPoint *(PFN) - input*
: Address of the external function entry point within the application EXE file. Functions registered with RexxRegisterFunctionExe are *local* to the current process. REXX procedures in the same process as the RexxRegisterFunctionExe issuer can call local external functions.

RexxRegisterFunctionExe - Returns

The RexxRegisterFunctionExe return values are:

0	RXFUNC_OK	The call to the function completed successfully.
10	RXFUNC_DEFINED	The requested function is already registered.
20	RXFUNC_NOMEM	There is not enough memory to register a new function.

RexxDeregisterFunction

RexxDeregisterFunction deregisters an external function.

Topics:

- Syntax
- Parameters
- Returns

RexxDeregisterFunction - Syntax

RexxDeregisterFunction(FuncName)

RexxDeregisterFunction - Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name to deregister.

RexxDeregisterFunction - Returns

The RexxDeregisterFunction return values are:

0	RXFUNC_OK	The call to the function completed successfully.
30	RXFUNC_NOTREG	The requested function is not registered.

RexxQueryFunction

RexxQueryFunction queries the existence of a registered external function.

Topics:

RexxQueryFunction - Syntax

RexxQueryFunction(FuncName)

RexxQueryFunction - Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name to query.

RexxQueryFunction - Returns

The RexxQueryFunction return values are:

0	RXFUNC_OK	The call to the function completed successfully.
30	RXFUNC_NOTREG	The requested function is not registered.

RexxQueryFunction - Remarks

RexxQueryFunction returns RXFUNC_OK only if the requested function is available to the current process. If a function is not available to the current process, RexxQueryFunction searches the RexxRegisterFunctionDll external function list.

External Functions Interface Returns

Return codes for the External Functions Interface are as follows:

0	RXFUNC_OK	The call to the function completed successfully.
10	RXFUNC_DEFINED	The requested function is already registered.
20	RXFUNC_NOMEM	There is not enough memory to register a new function.
30	RXFUNC_NOTREG	The requested function is not registered.
40	RXFUNC_MODNOTFND	The dynamic link library module could not be found.
50	RXFUNC_ENTNOFIND	The dynamic link library entry point could not be found.

System Exit Interface

The REXX System Exits let the programmer create a customized REXX operating environment. You can set up user-defined exit handlers to process specific REXX activities.

Applications can create exits for:

- The administration of resources at the beginning and end of interpretation
- Linkages to external functions and subcommand handlers
- Special language features; for example, input and output to standard resources
- Polling for halt and external trace events

Exit handlers are similar to subcommand handlers and external functions:

- Applications must register named exit handlers with the REXX interpreter.
- Exit handlers can reside in dynamic link libraries or within an application EXE module.

Writing System Exit Handlers

The following is a sample exit handler definition:

Sample system exit handler definition

```
LONG Rexx_IO_exit(
     LONG  ExitNumber,    /* code defining the exit function    */
     LONG  Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock);    /* function dependent control block   */
```

ExitNumber	The major function code defining the type of exit call.
Subfunction	The subfunction code defining the exit event for the call.
ParmBlock	A pointer to the exit parameter list. The exit parameter list contains exit-specific information. See the exit descriptions following parameter list formats. **Note:** Some exit subfunctions do not have parameters. *ParmBlock* is set to null for exit subfunctions without parameters.

Exit Return Codes

Exit handlers return an integer value that signals one of three actions :

RXEXIT_HANDLED
: The exit handler processed the exit subfunction and updated the subfunction parameter list as required. The REXX interpreter continues with processing as usual.

RXEXIT_NOT_HANDLED
: The exit handler did not process the exit subfunction. The REXX interpreter processes the subfunction as if the exit handler was not called.

RXEXIT_RAISE_ERROR
: A fatal error occurred in the exit handler. The REXX interpreter raises REXX error 48 ("Failure in system service").

For example, if an application creates an input/output exit handler:

- When the exit handler returns RXEXIT_NOT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter writes the output line to STDOUT.
- When the exit handler returns RXEXIT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter assumes the exit handler has handled all required output. The interpreter does not write the output line to STDOUT.
- When the exit handler returns RXEXIT_RAISE_ERROR for an RXSIOSAY subfunction, the interpreter raises REXX error 48, "Failure in system service".

Exit Parameters

Each exit subfunction has a different parameter list. All RXSTRING exit subfunction parameters are passed as null-terminated RXSTRINGs. The RXSTRING value may also contain null characters.

For some exit subfunctions, the exit handler may return an RXSTRING character result in the parameter list. The interpreter provides a default 256-byte RXSTRING for the result string. If the result is longer than 256 bytes, a new RXSTRING can be allocated using DosAllocMem. The REXX interpreter returns the RXSTRING storage for the exit handler.

Identifying Exit Handlers to REXX

System exit handlers must be registered with RexxRegisterExitDll or RexxRegisterExitExe. The system exit handler registration is similar to subcommand handler registration.

The REXX system exits are enabled with the RexxStart function parameter *Exits*. *Exits* is a pointer to an array of RXSYSEXIT structures. Each RXSYSEXIT structure in the array contains a REXX exit code and the address of an ASCIIZ exit handler name. The RXENDLST exit code marks the exit list end.

RXSYSEXIT data structure

```
typedef struct {
   PSZ             sysexit_name;       /* name of exit handler        */
   LONG            sysexit_code;       /* system exit function code   */
} RXSYSEXIT;
```

The REXX interpreter calls the registered exit handler named in *sysexit_name* for all of the *sysexit_code* subfunctions.

Example

The following is a sample of system exit usage:

Sample system exit usage

```
WORKAREARECORD  *user_info[2];         /* saved user information     */
RXSYSEXIT exit_list[2];               /* system exit list           */
  user_info[0] = global_workarea;     /* save global work area for  */
  user_info[1] = NULL;                /* re-entrancy                */
  rc = RexxRegisterExitExe("EditInit", /* register exit handler     */
      &Init_exit,                     /* located at this address    */
      user_info);                     /* save global pointer        */
                                      /* set up for RXINI exit      */
  exit_list[0].sysexit_name = "EditInit";
  exit_list[0].sysexit_code = RXINI;
  exit_list[1].sysexit_code = RXENDLST;
  return_code = RexxStart(1,          /* one argument               */
                          argv,       /* argument array             */
                          "CHANGE.ED", /* REXX procedure name       */
                          NULL,       /* use disk version           */
                          "Editor",   /* default address name       */
                          RXCOMMAND,  /* calling as a subcommand    */
                          exit_list,  /* exit list                  */
                          &rc,        /* converted return code      */
                          &retstr);   /* returned result            */
                                      /* process return value       */
.
.
.
}
LONG Init_exit(
     LONG  ExitNumber,    /* code defining the exit function    */
     LONG  Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock)     /* function dependent control block   */
{
  WORKAREARECORD  *user_info[2];      /* saved user information     */
  WORKAREARECORD   global_workarea;   /* application data anchor    */
  USHORT           query_flag;        /* flag for handler query     */
  rc = RexxQueryExit("EditInit",      /* retrieve application work  */
      NULL,                           /* area anchor from REXX.     */
      &query_flag,
      user_info);
  global_workarea = user_info[0];     /* set the global anchor      */
  if (global_workarea->rexx_trace)    /* trace at start?            */
                                      /* turn on macro tracing      */
    RexxSetTrace(global_workarea->rexx_pid, global_workarea->rexx_tid);
  return RXEXIT_HANDLED;              /* successfully handled       */
}
```

System Exit Definitions

The REXX interpreter supports the system exits listed below. Exit subfunctions link to more information, including:

- When REXX calls the exit
- The default action when the exit is not provided or the exit handler does not process the subfunction
- The exit action
- Applicable continuations
- The subfunction parameters

The system exits and their subfunctions are:

RXFNC is the external function call exit.

The variable pool interface is fully enabled during calls to this exit.

RXFNCCAL	calls an external function.

RXCMD is the subcommand call exit.

The variable pool interface is fully enabled during calls to these exits.

RXCMDHST	calls a subcommand handler.

RXMSQ is the external data queue exit.

The variable pool interface is enabled for RXSHV_EXIT requests during calls to these exits.

RXMSQPLL	pulls a line from the external data queue.
RXMSQPSH	places a line on the external data queue.
RXMSQSIZ	returns the number of lines on the external data queue.
RXMSQNAM	sets the active external data queue name.

RXSIO is the standard input and output exit.

The PARSE LINEIN instruction and the LINEIN, LINEOUT, LINES, CHARIN, CHAROUT, and CHARS built-in functions do not call the RXSIO exit.

RXSIOSAY	writes a line to the standard output stream for the SAY instruction.
RXSIOTRC	writes a line to the standard error stream for REXX trace or REXX error messages.
RXSIOTRD	reads a line from the standard input stream for PULL or PARSE PULL.
RXSIODTR	reads a line from the standard input stream for interactive debug.

RXHLT is the halt processing exit.

Because the RXHLT exit is called after every REXX instruction, enabling this exit slows REXX program execution. The RexxSetHalt function can halt a REXX program without between-instruction polling.

RXHLTTST	tests for a HALT condition.
RXHLTCLR	clears a HALT condition.

RXTRC is the external trace exit.

Because the RXTRC exit handler is called after every REXX instruction, enabling this exit slows REXX program execution. The RexxSetTrace function can turn on REXX tracing without the between-instruction polling.

RXTRCTST	tests for an external trace event.

RXINI is the initialization exit.

This exit is called as the last step of REXX program initialization. The variable pool interface is fully enabled during calls to this exit.

RXINIEXT	allows additional REXX procedure initialization.

RXTER is the termination exit.

This exit is called as the first step of REXX program termination. The variable pool interface is fully enabled during calls to this exit.

RXTEREXT processes REXX procedure termination.

RXFNCCAL

RXFNCCAL processes calls to external functions.

When called: When REXX calls an external subroutine or function.

Default action: Call the external routine using the usual external function search order.

Exit Action: Call the external routine, if possible.

Continuation: If necessary, raise REXX error 40 ("Incorrect call to routine"), 43 ("Routine not found"), or 44 ("Function or message did not return data").

RXFNCCAL Parameters

```
typedef struct {
   struct {
      unsigned rxfferr  : 1;          /* Invalid call to routine.    */
      unsigned rxffnfnd : 1;          /* Function not found.         */
      unsigned rxffsub  : 1;          /* Called as a subroutine if   */
                                      /* TRUE.  Return values are    */
                                      /* optional for subroutines,   */
                                      /* required for functions.     */
   } rxfnc_flags ;
   PUCHAR           rxfnc_name;       /* Pointer to function name.   */
   USHORT           rxfnc_namel;      /* Length of function name.    */
   PUCHAR           rxfnc_que;        /* Current queue name.         */
   USHORT           rxfnc_quel;       /* Length of queue name.       */
   USHORT           rxfnc_argc;       /* Number of args in list.     */
   PRXSTRING        rxfnc_argv;       /* Pointer to argument list.   */
                                      /* List mimics argv list for   */
                                      /* function calls, an array of */
                                      /* RXSTRINGs.                  */
   RXSTRING         rxfnc_retc;       /* Return value.               */
} RXFNCCAL_PARM;
```

The name of the external function is defined by *rxfnc_name* and *rxfnc_namel*. The arguments to the function are in *rxfnc_argc* and *rxfnc_argv*. If you call the named external function with the REXX CALL instruction (rather than using a function call), the flag *rxffsub* is TRUE.

The exit handler can set *rxfnc_flags* to indicate whether the external function call was successful. If neither *rxfferr* or *rxffnfnd* is TRUE, the exit handler successfully called the external function. The error flags are checked only when the exit handler handles the request.

The exit handler sets *rxffnfnd* to TRUE when the exit handler cannot locate the external function. The interpreter raises REXX error 43, "Routine not found". The exit handler sets *rxfferr* to TRUE when the exit handler locates the external function, but the external function returned an error return code. The REXX interpreter raises error 40, "Incorrect call to routine."

The exit handler returns the external function result in the *rxfnc_retc* RXSTRING. The REXX interpreter raises error 44, "Function or method did not return data," when the external routine is called as a function and the exit handler does not return a result. When the external routine

is called with the REXX CALL instruction, a result is not required.

RXCMDHST

RXCMDHST calls a named subcommand handler.

When called: When REXX procedure issues a command.

Default action: Call the named subcommand handler specified by the current REXX ADDRESS setting.

Exit Action: Process the call to a named subcommand handler.

Continuation: Raise the ERROR or FAILURE condition when indicated by the parameter list flags.

RXCMDHST Parameters

```
typedef struct {
   struct {                            /* Condition flags             */
      unsigned rxfcfail : 1;           /* Command failed.  Trap with  */
                                       /* CALL or SIGNAL on FAILURE.  */
      unsigned rxfcerr  : 1;           /* Command ERROR occurred.     */
                                       /* Trap with CALL or SIGNAL on */
                                       /* ERROR.                      */
   } rxcmd_flags;
   PUCHAR            rxcmd_address;    /* Pointer to address name.    */
   USHORT            rxcmd_addressl;   /* Length of address name.     */
   PUCHAR            rxcmd_dll;        /* dll name for command.       */
   USHORT            rxcmd_dll_len;    /* Length of dll name.  0 ==>  */
                                       /*.EXE file.                  */
   RXSTRING          rxcmd_command;    /* The command string.         */
   RXSTRING          rxcmd_retc;       /* Pointer to return code      */
                                       /* buffer.  User allocated.    */
} RXCMDHST_PARM;
```

The *rxcmd_command* field contains the issued command. *Rxcmd_address* , *rxcmd_addressl* , *rxcmd_dll* , and *rxcmd_dll_len* fully define the current ADDRESS setting. *Rxcmd_retc* is an RXSTRING for the return code value assigned to REXX special variable RC.

The exit handler can set *rxfcfail* or *rxfcerr* to TRUE to raise an ERROR or FAILURE condition.

RXMSQPLL

RXMSQPLL pulls a line from the external data queue.

When called: When a REXX PULL instruction, PARSE PULL instruction, or LINEIN built-in function reads a line from the external data queue.

Default action: Remove a line from the current REXX data queue.

Exit Action: Return a line from the data queue that the exit handler provided.

RXMSQPLL Parameters

```
typedef struct {
   RXSTRING          rxmsq_retc;       /* Pointer to dequeued entry   */
                                       /* buffer.  User allocated.    */
} RXMSQPLL_PARM;
```

The exit handler returns the queue line in the *rxmsq_retc* RXSTRING.

RXMSQPSH

RXMSQPSH places a line on the external data queue.

When called: Called by the REXX PUSH instruction, QUEUE instruction, or LINEOUT built-in function to add a line to the data queue.

Default action: Add the line to the current REXX data queue.

Exit Action: Add the line to the data queue that the exit handler provided.

RXMSQPSH Parameters

```
typedef struct {
   struct {                            /* Operation flag              */
      unsigned rxfmlifo : 1;           /* Stack entry LIFO when TRUE, */
                                       /* FIFO when FALSE.            */
   } rxmsq_flags;
   RXSTRING          rxmsq_value;      /* The entry to be pushed.     */
} RXMSQPSH_PARM;
```

The *rxmsq_value* RXSTRING contains the line added to the queue. It is the responsibility of the exit handler to truncate the string if the exit handler data queue has a maximum length restriction. *Rxfmlifo* is the stacking order (LIFO or FIFO).

RXMSQSIZ

RXMSQSIZ returns the number of lines in the external data queue.

When called: When the REXX QUEUED built-in function requests the size of the external data queue.

Default action: Request the size from the current REXX data queue.

Exit Action: Return the size of the data queue that the exit handler provided.

RXMSQSIZ Parameters

```
typedef struct {
   ULONG            rxmsq_size;       /* Number of Lines in Queue    */
} RXMSQSIZ_PARM;
```

The exit handler returns the number of queue lines in *rxmsq_size* .

RXMSQNAM

RXMSQNAM sets the name of the active external data queue.

When called: Called by the RXQUEUE("SET", *newname*) built-in function.

Default action: Change the current default queue to *newname* .

Exit Action: Change the default queue name for the data queue that the exit handler provided.

RXMSQNAM Parameters

```
typedef struct {
   RXSTRING         rxmsq_name;       /* RXSTRING containing         */
                                      /* queue name.                 */
} RXMSQNAM_PARM;
```

Rxmsq_name contains the new queue name.

RXSIOSAY

RXSIOSAY writes a line to the standard output stream.

When called: By the SAY instruction to write a line to the standard output stream.

Default action: Write to the standard output stream (STDOUT).

Exit Action: Write the line to the output stream that the exit handler provided.

RXSIOSAY Parameters

```
typedef struct {
   RXSTRING          rxsio_string;     /* String to display.          */
} RXSIOSAY_PARM;
```

The output line is contained in *rxsio_string* . The output line can be any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRC

RXSIOTRC writes trace and error message output to the standard error stream.

When called: To output lines of trace output and REXX error messages.

Default action: Write the line to the standard error stream (.ERROR).

Exit Action: Write the line to the error output stream that the exit handler provided.

RXSIOTRC Parameters

```
typedef struct {
  RXSTRING           rxsio_string;     /* Trace line to display.      */
} RXSIOTRC_PARM;
```

The output line is contained in *rxsio_string* . The output line may be of any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRD

RXSIOTRD reads from a standard input stream.

When called: To read from the standard input stream for the REXX PULL and PARSE PULL instructions.

Default action: Read a line from the standard input stream (STDIN).

Exit Action: Return a line from the standard input stream that the exit handler provided.

RXSIOTRD Parameters

```
typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.        */
} RXSIOTRD_PARM;
```

The input stream line is returned in the *rxsiotrd_retc* RXSTRING.

RXSIODTR

RXSIODTR reads the interactive debug input.

When called: To read from the debug input stream for interactive debug prompts.

Default action: Read a line from the standard input stream (STDIN).

Exit Action: Return a line from the standard debug stream that the exit handler provided.

RXSIODTR Parameters

```
typedef struct {
   RXSTRING          rxsiodtr_retc;    /* RXSTRING for output.        */
} RXSIODTR_PARM;
```

The input stream line is returned in the *rxsiodtr_retc* RXSTRING.

RXHLTTST

RXHLTTST tests the HALT indicator.

When called: When the interpreter polls externally raised HALT conditions. The exit will be called after completion of every

REXX instruction.

Default action: The interpreter uses the system facilities for trapping Cntrl-Break signals.

Exit Action: Return the current state of the HALT condition (either TRUE or FALSE).

Continuation: Raise the REXX HALT condition if the exit handler returns TRUE.

RXHLTTST Parameters

```
typedef struct {
   struct {                          /* Halt flag                  */
      unsigned rxfhhalt : 1;         /* Set if HALT occurred.      */
   } rxhlt_flags;
} RXHLTTST_PARM;
```

If the exit handler sets *rxfhhalt* to TRUE, the HALT condition is raised in the REXX program.

When the exit handler has set *rxfhhalt* to TRUE, it can also use the RXSHV_EXIT operation of RexxVariablePool to return a string describing the HALT condition reason. The REXX program can retrieve the reason string using the CONDITION("D") built-in function.

RXHLTCLR

RXHLTCLR clears a HALT condition.

When called: To acknowledge processing of the HALT condition when the interpreter has recognized and raised a HALT condition

Default action: The interpreter resets the Cntrl-Break signal handlers.

Exit Action: Reset exit handler HALT state to FALSE.

RXHLTCLR Parameters

None.

RXTRCTST

RXTRCTST tests for an external trace indicator.

When called: When the interpreter polls for an external trace event. The exit is called after completion of every REXX instruction.

Default action: None.

Exit Action: Return the current state of external tracing (either TRUE or FALSE).

Continuation: When the exit handler switches from FALSE to TRUE, the REXX interpreter enters REXX interactive debug mode using TRACE ?R level of tracing. When the exit handler switches from TRUE to FALSE, the REXX interpreter exits interactive debug mode.

RXTRCTST Parameters

```
typedef struct {
  struct {
    unsigned rxftrace : 1;        /* External trace setting        */
  } rxtrc_flags;
} RXTRCTST_PARM;
```

If the exit handler switches *rxftrace* to TRUE, REXX switches on interactive debug mode. It the exit handler switches *rxftrace* to FALSE, REXX switches off interactive debug mode.

RXINIEXT

RXINIEXT allows for additional initialization processing.

When called: Before the first instruction of the REXX procedure is interpreted.

Default action: None.

Exit Action: The exit handler may perform additional initialization. For example:

- Use RexxVariablePool to initialize application specific variables
- Use RexxSetTrace to switch on REXX interactive debug mode.

RXINIEXT Parameters

None.

RXTEREXT

RXTEREXT performs termination processing.

When called: After the last instruction of the REXX procedure has been interpreted.

Default action: None.

Exit Action: The exit handler may perform additional termination activities. For example, the exit handler can use RexxVariablePool to retrieve REXX variable values.

RXTEREXT Parameters

None.

System Exit Functions

The system exit functions are similar to the subcommand handler functions. The system exit functions are:

RexxRegisterExitDll
RexxRegisterExitExe
RexxDeregisterExit
RexxQueryExit

RexxRegisterExitDll

RexxRegisterExitDll registers an exit handler that resides in a dynalink library routine.

Topics:

- Syntax
- Parameters
- Returns
- Remarks

RexxRegisterExitDll - Syntax

RexxRegisterExitDll(ExitName, ModuleName, EntryPoint, UserArea, DropAuth)

RexxRegisterExitDll - Parameters

ExitName *(PSZ) - input*
Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the exit handler routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynalink procedure name. *EntryPoint* is the routine within *ModuleName* that REXX calls as an exit handler.

UserArea *(PUCHAR) - input*
Address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the exit handler registration. *UserArea* can be NULL if there is no user information to save. The RexxQueryExit function can retrieve the saved user information.

DropAuth *(ULONG) - input*
The drop authority. *DropAuth* identifies the processes that can deregister the exit handler. The possible *DropAuth* values are:

RXEXIT_DROPPABLE
Any process can deregister the exit handler with RexxDeregisterExit.

RXEXIT_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterExit.

RexxRegisterExitDll - Returns

The RexxRegisterExitDll return values are:

0	RXEXIT_OK	Function completed successfully.
10	RXEXIT_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this exit handler you must specify its library name.)
1002	RXEXIT_NOEMEM	There is insufficient memory to complete this request.

RexxRegisterExitDll - Remarks

EntryPoint can be either a 16-bit or a 32-bit routine. REXX calls the exit handler in the correct addressing mode.

RexxRegisterExitExe

RexxRegisterExitExe registers an exit handler that resides within application code.

Topics:

RexxRegisterExitExe - Syntax

RexxRegisterExitExe(ExitName, EntryPoint, UserArea)

RexxRegisterExitExe - Parameters

ExitName *(PSZ) - input*
: Address of an ASCIIZ exit handler name.

EntryPoint *(PFN) - input*
: Address of the exit handler entry point within the application EXE file.

UserArea *(PUCHAR) - input*
: Address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the exit handler registration. *UserArea* can be NULL if there is no user information to save. The RexxQueryExit function can retrieve the user information.

RexxRegisterExitExe - Returns

The RexxRegisterExitExe return values are:

0	RXEXIT_OK	Function completed successfully.
10	RXEXIT_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this exit handler you must specify its library name.)
30	RXEXIT_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names

```
                                (RexxRegisterExitExe or
                                RexxRegisterExitDll); the exit
                                handler is not registered (other
                                REXX exit handler functions).

 1002   RXEXIT_NOEMEM           There is insufficient memory to
                                complete this request.
```

RexxRegisterExitExe - Remarks

If *ExitName* has the same name as a handler registered with RexxRegisterExitDll, RexxRegisterExitExe returns RXEXIT_DUP. This is not an error and the new exit handler has been properly registered.

RexxRegisterExitExe - Example

The following is a sample exit handler registration:

Sample exit handler registration

```
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  user_info[0] = global_workarea;      /* save global work area for  */
  user_info[1] = NULL;                 /* re-entrancy                */
  rc = RexxRegisterExitExe("IO_Exit",  /* register editor handler    */
      &Edit_IO_Exit,                   /* located at this address    */
      user_info);                      /* save global pointer        */
```

RexxDeregisterExit

RexxDeregisterExit deregisters an exit handler.

Topics:

RexxDeregisterExit - Syntax

RexxDeregisterExit(ExitName, ModuleName)

RexxDeregisterExit - Parameters

ExitName *(PSZ) - input*
: Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
: Address of an ASCIIZ dynamic link library name. *ModuleName* restricts the query to an exit handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxDeregisterExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxDeregisterExit does not find a RexxRegisterExitExe handler, it searches the RexxRegisterExitDll exit handler list.

RexxDeregisterExit - Returns

The RexxDeregisterExit return values are:

0	RXEXIT_OK	Function completed successfully.
30	RXEXIT_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterExitExe or RexxRegisterExitDll); the exit handler is not registered (other REXX exit handler functions).
40	RXEXIT_NOCANDROP	The exit handler has been registered as "not droppable."

RexxDeregisterExit - Remarks

The handler is removed from the exit handler list.

RexxQueryExit

RexxQueryExit queries an exit handler and retrieves saved user information.

Topics

- Example

RexxQueryExit - Syntax

RexxQueryExit(ExitName, ModuleName, Flag, UserWord)

RexxQueryExit - Parameters

ExitName *(PSZ) - input*
: Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
: *ModuleName* restricts the query to an exit handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxQueryExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxQueryExit does not find a RexxRegisterExitExe handler, it searches the RexxRegisterExitDll exit handler list.

Flag *(PUSHORT) - output*
: Exit handler registration flag. *Flag* is the *ExitName* exit handler registration status. When RexxQueryExit returns RXEXIT_OK, the *ExitName* exit handler is currently registered. When RexxQueryExit returns RXEXIT_NOTREG, the *ExitName* exit handler is not registered.

UserWord *(PUCHAR) - output*
: Address of an 8-byte area to receive the user information saved with RexxRegisterExitExe or RexxRegisterExitDll. *UserWord* can be NULL if the saved user information is not required.

RexxQueryExit - Returns

The RexxQueryExit return values are:

```
0      RXEXIT_OK                Function completed successfully.

30     RXEXIT_NOTREG            Registration was unsuccessful due to
                                duplicate handler and dynalink names
                                ( RexxRegisterExitExe or
                                RexxRegisterExitDll); the exit
                                handler is not registered (other
                                REXX exit handler functions).
```

RexxQueryExit - Example

The following is a sample exit handler query:

Sample exit handler query

```
ULONG Edit_IO_Exit(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer to short for return of flags     */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */
  rc = RexxQueryExit("IO_Exit",        /* retrieve application work  */
      NULL,                            /* area anchor from REXX.     */
      &query_flag,
      user_info);
  global_workarea = user_info[0];      /* set the global anchor      */
```

System Exit Interface Returns

The return codes for the System Exit Interface are as follows:

0	RXEXIT_OK	Function completed successfully.
10	RXEXIT_DUP	A duplicate handler name has been successfully registered. There is either an EXE handler with the same name registered in another process, or a DLL handler with the same name registered in another DLL. (To address this exit handler you must specify its library name.)
30	RXEXIT_NOTREG	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterExitExe or RexxRegisterExitDll); the exit handler is not registered (other REXX exit handler functions).
40	RXEXIT_NOCANDROP	The exit handler has been registered as "not droppable."
1002	RXEXIT_NOEMEM	There is insufficient memory to complete this request.

Variable Pool Interface

Application programs can use the REXX Variable Pool Interface to manipulate the variables of a currently active REXX procedure.

Interface Types

Three of the Variable Pool Interface functions (set, fetch, and drop) have dual interfaces.

Symbolic Interface

The symbolic interface uses normal REXX variable rules when interpreting variables. Variable names are valid REXX symbols (in mixed case if desired) including compound symbols. Compound symbols are referenced with tail substitution. The functions that use the symbolic interface are RXSHV_SYSET, RXSHV_SYFET, and RXSHV_SYDRO.

Direct Interface

The direct interface uses no substitution or case translation. Simple symbols must be valid REXX variable names. A valid REXX variable name:

- Does not begin with a digit or period
- Contains only uppercase A to Z, the digits 0 - 9, or the characters _, ! or ? before the first period of the name.
- Can contain any characters after the first period of the name.

Compound variables are specified using the derived name of the variable. Any characters (including blanks) may appear after the first period of the name. No additional variable substitution is used. RXSHV_SET, RXSHV_FETCH, and RXSHV_DROP use the direct interface.

RexxVariablePool Restrictions

Only the main thread of an application can access the REXX variable pool. Applications may create and use new threads, but only the original thread that called RexxStart can use RexxVariablePool.

OS/2 EXE modules called from a REXX procedure execute in a new process. Because the modules are not using the same process and thread as the REXX procedure, the modules cannot use RexxVariablePool to access REXX variables. You can use RexxVariablePool from subcommand handlers, external functions and exit handlers.

RexxVariablePool Interface Function

REXX procedure variables are accessed using the RexxVariablePool function.

RexxVariablePool

RexxVariablePool accesses variables of a currently active REXX procedure.

Topics:

- Syntax
- Parameters
- Returns

- Example

RexxVariablePool - Syntax

RexxVariablePool(RequestBlockList)

RexxVariablePool - Parameters

RequestBlockList *(PSHVBLOCK) - input*
A linked list of shared variable request blocks (SHVBLOCK). Each shared variable request block in the linked list is a separate variable access request.

The SHVBLOCK has the following form:

SHVBLOCK data structure

```
typedef struct shvnode {
    struct shvnode    *shvnext;
    RXSTRING           shvname;
    RXSTRING           shvvalue;
    ULONG              shvnamelen;
    ULONG              shvvaluelen;
    UCHAR              shvcode;
    UCHAR              shvret;
}   SHVBLOCK;
```

shvnext — The address of the next SHVBLOCK in the request list. *shvnext* is NULL for the last request block.

shvname — An RXSTRING containing a REXX variable name. *shvname* usage varies for the different SHVBLOCK request codes:

RXSHV_SET

RXSHV_SYSET

RXSHV_FETCH

RXSHV_SYFET

RXSHV_DROPV

RXSHV_SYDRO

RXSHV_PRIV — *shvname* is an RXSTRING pointing to the name of the REXX variable the shared variable request block accesses.

RXSHV_NEXTV — *shvname* is an RXSTRING defining an area of storage to receive the name of the next variable. *shvnamelen* is the length of the RXSTRING area. If the variable name is longer than *shvnamelen* characters, the name is truncated and the RXSHV_TRUNC bit of *shvret* is set. On return, *shvname.strlength* contains the length of the variable name; *shvnamelen* is unchanged.

If *shvname* is an empty RXSTRING (*strptr* is NULL), the REXX interpreter allocates and returns an RXSTRING to hold the variable name. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur.

However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory is not allocated for that request block. The RexxVariablePool caller must release the storage with DosFreeMem.

Note: The RexxVariablePool does not add a terminating null character to the variable name.

RXSHV_EXIT — *shvname* is unused for the RXSHV_EXIT function.

shvvalue — An RXSTRING containing a REXX variable value. The meaning of *shvvalue* varies for the different SHVBLOCK request codes:

RXSHV_SET

RXSHV_SYSET — *shvvalue* is the value assigned to the REXX variable in *shvname* . *shvvaluelen* contains the length of the variable value.

RXSHV_EXIT — *shvvalue* is the value assigned to the exit handler return value. *shvvaluelen* contains the length of the variable value.

RXSHV_FETCH

RXSHV_SYFET

RXSHV_PRIV

RXSHV_NEXT — *shvvalue* is a buffer the REXX interpreter uses to return a copy of REXX variable *shvname* . *shvvaluelen* contains the length of the value buffer. On return, *shvvalue.strlength* is set to the length of the returned value and *shvvaluelen* is unchanged. If the variable value is longer than *shvvaluelen* characters, the value is truncated and the RXSHV_TRUNC bit of *shvret* is set. On return, *shvvalue.strlength* is set to the length of the returned value; *shvvaluelen* is unchanged.

If *shvvalue* is an empty RXSTRING (*strptr* is NULL), the REXX interpreter allocates and returns an RXSTRING to hold the variable value. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur. However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory is not allocated for that request block. The RexxVariablePool caller must release the storage with DosFreeMem.

Note: The RexxVariablePool does not add a terminating null character to the variable value.

RXSHV_DROPV

RXSHV_SYDRO — *shvvalue* is not used.

shvcode — The shared variable block request code. The request codes are:

RXSHV_SET

RXSHV_SYSET — Assign a new value to a REXX procedure variable.

RXSHV_FETCH

RXSHV_SYFET — Retrieve the value of a REXX procedure variable.

RXSHV_DROPV

RXSHV_SYDRO — Drop (unassign) a REXX procedure variable.

RXSHV_PRIV — Fetch REXX procedure private information. The following information items can be retrieved by name:

EXITNAME

The name of the current system exit handler for this thread. If not called from within an exit handler, a null string will be returned.

PARM

The number of arguments supplied to the REXX procedure. The number is formatted as a character string.

PARM.n

The Nth argument string to the REXX procedure. If the Nth argument was not supplied to the procedure (either omitted or fewer than N parameters were specified), a null string is returned.

QUENAME

The current REXX data queue name.

SOURCE

The REXX procedure source string used for the PARSE SOURCE instruction.

VERSION

The REXX interpreter version string used for the PARSE SOURCE instruction.

RXSHV_NEXTV

Fetch next variable. RXSHV_NEXTV traverses the variables in the current generation of REXX variables, excluding variables hidden by PROCEDURE instructions. The variables are not returned in any specified order.

The REXX interpreter maintains an internal pointer to its list of variables. The variable pointer is reset to the first REXX variable whenever:

1. An external program returns control to the interpreter
2. A set, fetch or drop RexxVariablePool function is used.

RXSHV_NEXTV returns both the name and the value of REXX variables until the end of the variable list is reached. If no REXX variables are left to return, RexxVariablePool sets the RXSHV_LVAR bit in *shvret*.

RXSHV_EXIT

Set a return value for an external function or system exit call. RXSHV_EXIT is valid only from external functions or system exit events that return a string value. An external function or exit handler can use RXSHV_EXIT only once.

shvret

Individual shared variable request return code. *shvret* is a 1-byte field of status flags for the individual shared variable request. The *shvret* fields for all request blocks in the list are ORed together to form the RexxVariablePool return code. The individual status conditions are:

RXSHV_OK

The request was processed without error (all flag bits are FALSE).

RXSHV_NEWV

The named variable was uninitialized at the time of the call.

RXSHV_LVAR

No more variables are available for an RXSHV_NEXTV operation.

RXSHV_TRUNC

A variable value or variable name was truncated because the supplied RXSTRING was too small for the copied value.

RXSHV_BADN

The variable name specified in *shvname* was invalid for the requested operation.

RXSHV_MEMFL

The REXX interpreter was unable to obtain the storage required to complete the request.

RXSHV_BADF

The shared variable request block contains an invalid function code.

The REXX interpreter processes each request block in the order provided; RexxVariablePool returns to the caller after the last block is processed or after a severe error (such as an out-of-memory condition).

The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The return codes for all of the individual requests are ORed together to form the composite return code. Individual shared variable request return codes are returned in the request shared variable blocks.

RexxVariablePool - Returns

The RexxVariablePool return values are:

0 to 127
: RexxVariablePool has processed the entire shared variable request block list.

  The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The low-order 6 bits of the *shvret* fields for all request blocks are ORed together to form the composite return code. Individual shared variable request status flags are returned in the shared variable request block *shvret* field.

RXSHV_NOAVL
: The variable pool interface was not enabled when call was issued.

RexxVariablePool - Example

The following is a sample call to RexxVariablePool:

Sample call to RexxVariablePool

```
/*********************************************************************/
/*                                                                   */
/* SetRexxVariable - Set the value of a REXX variable                */
/*                                                                   */
/*********************************************************************/
INT SetRexxVariable(
  PSZ        name,                   /* REXX variable to set       */
  PSZ        value)                  /* value to assign            */
{
  SHVBLOCK   block;                  /* variable pool control block*/
  block.shvcode = RXSHV_SYSET;       /* do a symbolic set operation*/
  block.shvret=(UCHAR)0;             /* clear return code field    */
  block.shvnext=(PSHVBLOCK)0;        /* no next block              */
                                     /* set variable name string   */
  MAKERXSTRING(block.shvname, name, strlen(name));
                                     /* set value string           */
  MAKERXSTRING(block.shvvalue, value, strlen(value));
  block.shvvaluelen=strlen(value);   /* set value length           */
  return RexxVariablePool(&block);   /* set the variable           */
}
```

Queue Interface

Application programs can use the REXX Queue Interface to establish and manipulate named queues. Named queues prevent different REXX programs that are running in a single session from interfering with each other. Named queues also allow REXX programs running in different sessions to synchronize execution and pass data. These queuing services are entirely separate from the OS/2 Inter-Process Communications queues.

Note: See the *Object REXX Reference* for more information about named and unnamed queues.

Queue Interface Functions

The functions for creating and using named queues are:

RexxCreateQueue
RexxDeleteQueue
RexxQueryQueue
RexxAddQueue
RexxPullQueue

RexxCreateQueue

RexxCreateQueue creates a new (empty) queue.

Topics:

- Syntax
- Parameters
- Returns
- Remarks

RexxCreateQueue - Syntax

RexxCreateQueue(Buffer, BuffLen, RequestedName, DupFlag)

RexxCreateQueue - Parameters

Buffer *(PSZ) - input*
: Address of the buffer where the ASCIIZ name of the created queue is returned.

BuffLen *(ULONG) - input*
: Size of the buffer.

RequestedName *(PSZ) - input*
: Address of an ASCIIZ queue name. If no queue of that name already exists, a queue is created with the requested name. If the name already exists, a queue is still created; however, REXX chooses an arbitrary name for the queue. In addition, the *DupFlag* is set.

  When *RequestedName* is NULL, REXX provides a name for the created queue.

In all cases, the actual queue name is passed back to the caller.

DupFlag *(PULONG) - output*
Duplicate name indicator. This flag is set when the requested name already exists.

RexxCreateQueue - Returns

The RexxCreateQueue return values are:

0	RXQUEUE_OK	The queue function has completed successfully.
1	RXQUEUE_STORAGE	The name buffer is not large enough for the queue name.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."

RexxCreateQueue - Remarks

Queue names must conform to the same syntax rules as for REXX variable names. Lower case characters in queue names are translated to upper case.

The queue name must be a valid REXX symbol. However, there is no connection between queue names and variable names. A program can have a variable and a queue with a common name.

RexxDeleteQueue

RexxDeleteQueue deletes a queue.

Topics:

RexxDeleteQueue - Syntax

RexxDeleteQueue(QueueName)

RexxDeleteQueue - Parameters

QueueName *(PSZ) - input*
Address of the ASCIIZ name of the queue to be deleted.

RexxDeleteQueue - Returns

The RexxDeleteQueue return values are:

0	RXQUEUE_OK	The queue function has completed successfully.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."
9	RXQUEUE_NOTREG	The queue does not exist.
10	RXQUEUE_ACCESS	The queue cannot be deleted because it is busy.

RexxDeleteQueue - Remarks

If a queue is busy (for example, wait is active), it is not deleted.

RexxQueryQueue

RexxQueryQueue returns the number of entries remaining in the named queue.

Topics:

- Syntax
- Parameters
- Returns

RexxQueryQueue - Syntax

RexxQueryQueue(QueueName, Count)

RexxQueryQueue - Parameters

QueueName *(PSZ) - input*
Address of the ASCIIZ name of the queue to query.

Count *(PULONG) - output*
Number of entries in the queue.

RexxQueryQueue - Returns

The RexxQueryQueue return values are:

0	RXQUEUE_OK	The queue function has completed successfully.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."
9	RXQUEUE_NOTREG	The queue does not exist.

RexxAddQueue

RexxAddQueue adds an entry into a queue.

Topics:

RexxAddQueue - Syntax

RexxAddQueue(QueueName, EntryData, AddFlag)

RexxAddQueue - Parameters

QueueName *(PSZ) - input*
Address of the ASCIIZ name of the queue to which data is to be added.

EntryData *(PRXSTRING) - input*
Address of an RXSTRING containing the data to be added to the queue.

AddFlag *(ULONG) - input*
LIFO/FIFO flag. When *AddFlag* is `RXQUEUE_LIFO`, data is added LIFO (Last In, First Out) to the queue. When *AddFlag* is `RXQUEUE_FIFO`, data is added FIFO (First In, First Out).

RexxAddQueue - Returns

The RexxAddQueue return values are:

0	RXQUEUE_OK	The queue function has completed successfully.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."
6	RXQUEUE_PRIORITY	The order flag is not equal to RXQUEUE_LIFO or RXQUEUE_FIFO.
9	RXQUEUE_NOTREG	The queue does not exist.
12	RXQUEUE_MEMFAIL	There is insufficient memory available to complete the request.

RexxPullQueue

RexxPullQueue removes the top entry from the queue and returns it to the caller.

Topics:

RexxPullQueue - Syntax

RexxPullQueue(QueueName, DataBuf, DateTime, WaitFlag)

RexxPullQueue - Parameters

QueueName *(PSZ) - input*
: Address of the ASCIIZ name of the queue from which data is to be pulled.

DataBuf *(PRXSTRING) - output*
: Address of an RXSTRING for the returned value.

DateTime *(PDATETIME) - output*
: Address of the entry's date/time stamp.

WaitFlag *(ULONG) - input*
: Wait flag. When *WaitFlag* is `RXQUEUE_NOWAIT` and the queue is empty, RXQUEUE_EMPTY is returned. Otherwise, that is, *WaitFlag* is `RXQUEUE_WAIT`, REXX waits until a queue entry is available and returns that entry to the caller.

RexxPullQueue - Returns

The RexxPullQueue return values are:

0	RXQUEUE_OK	The queue function has completed successfully.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."
7	RXQUEUE_BADWAITFLAG	The wait flag is not equal to RXQUEUE_WAIT or RXQUEUE_NOWAIT.
8	RXQUEUE_EMPTY	Attempted to pull the item off the queue but it was empty.
9	RXQUEUE_NOTREG	The queue does not exist.
12	RXQUEUE_MEMFAIL	There is insufficient memory available to complete the

request.

RexxPullQueue - Remarks

The caller is responsible for freeing the returned memory pointed to by *DataBuf* .

Queue Interface Returns

The return codes for the Queue Interface are as follows:

0	RXQUEUE_OK	The queue function has completed successfully.
1	RXQUEUE_STORAGE	The name buffer is not large enough for the queue name.
5	RXQUEUE_BADQNAME	The queue name is not valid, or you tried to create or delete a queue named "SESSION."
6	RXQUEUE_PRIORITY	The order flag is not equal to RXQUEUE_LIFO or RXQUEUE_FIFO.
7	RXQUEUE_BADWAITFLAG	The wait flag is not equal to RXQUEUE_WAIT or RXQUEUE_NOWAIT.
8	RXQUEUE_EMPTY	Attempted to pull the item off the queue but it was empty.
9	RXQUEUE_NOTREG	The queue does not exist.
10	RXQUEUE_ACCESS	The queue cannot be deleted because it is busy.
12	RXQUEUE_MEMFAIL	There is insufficient memory available to complete the request.

Halt and Trace Interface

The halt and trace functions raise a REXX HALT condition or change the REXX interactive debug mode while a REXX procedure is running. These interfaces may be preferred over the RXHLT and RXTRC system exits. The system exits require an additional call to an exit routine after each REXX instruction completes. This might cause a noticeable performance degradation. The halt and trace functions are a single request to change the halt or trace state and do not degrade the REXX procedure performance.

Halt and Trace Interface Functions

The Halt and Trace functions include:

RexxSetHalt
RexxSetTrace
RexxResetTrace

RexxSetHalt

RexxSetHalt raises a HALT condition in a running REXX program.

Topics:

- Syntax
- Parameters
- Returns
- Remarks

RexxSetHalt - Syntax

RexxSetHalt(ProcessId, ThreadId)

RexxSetHalt - Parameters

ProcessId *(PID) - input*
: The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
: The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

RexxSetHalt - Returns

The RexxSetHalt return values are:

0	RXARI_OK	The function completed successfully.

1	RXARI_NOT_FOUND	The target REXX procedure was not found.
2	RXARI_PROCESSING_ERROR	A failure in REXX processing occurred.

RexxSetHalt - Remarks

This call is not processed if the target REXX program is running with the RXHLT exit enabled.

RexxSetTrace

RexxSetTrace turns on interactive debug mode for a REXX procedure.

Topics:

RexxSetTrace - Syntax

RexxSetTrace(ProcessId, ThreadId)

RexxSetTrace - Parameters

ProcessId *(PID) - input*
: The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
: The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

RexxSetTrace - Returns

The RexxSetTrace return values are:

0	RXARI_OK	The function completed successfully.
1	RXARI_NOT_FOUND	The target REXX procedure was not found.
2	RXARI_PROCESSING_ERROR	A failure in REXX processing occurred.

RexxSetTrace - Remarks

A RexxSetTrace call is not processed if the REXX procedure is using the RXTRC exit.

RexxResetTrace

RexxResetTrace turns off interactive debug mode for a REXX procedure.

Topics:

RexxResetTrace - Syntax

RexxResetTrace(ProcessId,ThreadId)

RexxResetTrace - Parameters

ProcessId *(PID) - input*

The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*

The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

RexxResetTrace - Returns

The RexxResetTrace return values are:

0	RXARI_OK	The function completed successfully.
1	RXARI_NOT_FOUND	The target REXX procedure was not found.
2	RXARI_PROCESSING_ERROR	A failure in REXX processing occurred.

RexxResetTrace - Remarks

A RexxResetTrace call is not processed if the REXX procedure is using the RXTRC exit.

Interactive debug is not turned off unless interactive debug mode was originally started with RexxSetTrace.

Halt and Trace Returns

The return codes for the halt and trace functions are as follows:

0	RXARI_OK	The function completed successfully.
1	RXARI_NOT_FOUND	The target REXX procedure was not found.
2	RXARI_PROCESSING_ERROR	A failure in REXX processing occurred.

Macrospace Interface

The macrospace can improve the performance of REXX procedures by maintaining REXX procedure images in memory for immediate load and execution. This is useful for frequently-used procedures and functions such as editor macros.

Programs registered in the REXX macrospace are available to all processes. You can run them by using the RexxStart function or calling them as functions or subroutines from other REXX procedures.

Procedures in the macrospace are called the same way other REXX external functions are called. However, the macrospace REXX procedures can be placed at the front or at the very end of the external function search order.

REXX procedures in the macrospace can be saved to a disk file. A saved macrospace file can be reloaded with a single call to RexxLoadMacroSpace. An application, such as an editor, can create its own library of frequently-used functions and load the entire library into memory for fast access. Multiple macrospace libraries may be created and loaded.

Search Order

When RexxAddMacro loads a REXX procedure into the macrospace, the position in the external function search order is specified. The REXX procedure may be placed before all other forms of external function or after all other external functions.

RXMACRO_SEARCH_BEFORE The REXX interpreter locates a function registered with RXMACRO_SEARCH_BEFORE before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER The REXX interpreter locates a function registered with RXMACRO_SEARCH_AFTER after any registered functions or external REXX files.

Storage of Macrospace Libraries

The REXX macrospace is placed in shared memory. Only the amount of memory and swap space available to the system limits the size of the macrospace. However, as the macrospace grows, it limits the memory available to other processes in the system. Allowing the macrospace to grow too large may degrade overall system performance due to increased system swap file access. Try to place only the most frequently-used functions in the macrospace.

Macrospace Interface Functions

The functions to manipulate macrospaces are:

RexxAddMacro
RexxDropMacro
RexxClearMacroSpace
RexxSaveMacroSpace
RexxLoadMacroSpace
RexxQueryMacro
RexxReorderMacro

RexxAddMacro

RexxAddMacro loads a REXX procedure into the macrospace.

Topics:

- Syntax

RexxAddMacro - Syntax

RexxAddMacro(FuncName, SourceFile, Position)

RexxAddMacro - Parameters

FuncName *(PSZ) - input*
: Address of the ASCIIZ function name. REXX procedures in the macrospace are called using the assigned function name.

SourceFile *(PSZ) - input*
: Address of the ASCIIZ file specification for the REXX procedure source file. When a file extension is not supplied,.CMD is used. When the full path is not specified, the current directory and path is searched.

Position *(ULONG) - input*
: Position in the REXX external function search order. Possible values are:

  RXMACRO_SEARCH_BEFORE
  : The REXX interpreter locates the function before any registered functions or external REXX files.

  RXMACRO_SEARCH_AFTER
  : The REXX interpreter locates the function after any registered functions or external REXX files.

RexxAddMacro - Returns

The RexxAddMacro return values are:

0	RXMACRO_OK	The call to the function completed successfully.
1	RXMACRO_NO_STORAGE	There was not enough memory to complete the requested function.
7	RXMACRO_SOURCE_NOT_FOUND	The requested file was not found.
8	RXMACRO_INVALID_POSITION	An invalid search-order position request flag was used.

RexxDropMacro

RexxDropMacro removes a REXX procedure from the macrospace.

Topics:

RexxDropMacro - Syntax

RexxDropMacro(FuncName)

RexxDropMacro - Parameters

FuncName *(PSZ) - input*
Address of the ASCIIZ function name.

RexxDropMacro - Returns

The RexxDropMacro return values are:

0	RXMACRO_OK	The call to the function completed successfully.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.

RexxClearMacroSpace

RexxClearMacroSpace removes all loaded REXX procedures from the macrospace.

Topics:

RexxClearMacroSpace - Syntax

RexxClearMacroSpace()

RexxClearMacroSpace - Returns

The RexxClearMacroSpace return values are:

0	RXMACRO_OK	The call to the function completed successfully.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.

RexxClearMacroSpace - Remarks

RexxClearMacroSpace must be used with care. This function removes all functions from the macrospace, including functions loaded by other processes.

RexxSaveMacroSpace

RexxSaveMacroSpace saves all or part of the macrospace REXX procedures to a disk file.

Topics:

RexxSaveMacroSpace - Syntax

RexxSaveMacroSpace(FuncCount, FuncNames, MacroLibFile)

RexxSaveMacroSpace - Parameters

FuncCount *(ULONG) - input*
: Number of REXX procedures to save.

FuncNames *(PSZ *) - input*
: Address of a list of ASCIIZ function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
: Address of the ASCIIZ macrospace file name. If *MacroLibFile* already exists, it is replaced with the new file.

RexxSaveMacroSpace - Returns

The RexxSaveMacroSpace return values are:

0	RXMACRO_OK	The call to the function completed successfully.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.
3	RXMACRO_EXTENSION_REQUIRED	An extension is required for the macrospace file name.
5	RXMACRO_FILE_ERROR	An error occurred accessing a macrospace file.

RexxSaveMacroSpace - Remarks

When *FuncCount* is zero or *FuncNames* is NULL, RexxSaveMacroSpace saves all functions in the macrospace.

Saved macrospace files can be used only with the same interpreter version that created the images. If RexxLoadMacroSpace is called to load a saved macrospace and the release level or service level is incorrect, RexxLoadMacroSpace fails. If RexxLoadMacroSpace fails, the REXX procedures must be reloaded individually from the original source programs.

RexxLoadMacroSpace

RexxLoadMacroSpace loads all or part of the REXX procedures from a saved macrospace file.

Topics:

RexxLoadMacroSpace - Syntax

RexxLoadMacroSpace(FuncCount, FuncNames, MacroLibFile)

RexxLoadMacroSpace - Parameters

FuncCount *(ULONG) - input*
: Number of REXX procedures to load from the saved macrospace.

FuncNames *(PSZ *) - input*
: Address of a list of ASCIIZ REXX function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
: Address of the ASCIIZ saved macrospace file name.

RexxLoadMacroSpace - Returns

The RexxLoadMacroSpace return values are:

```
0       RXMACRO_OK                      The call to the function
                                        completed successfully.

1       RXMACRO_NO_STORAGE              There was not enough
                                        memory to complete the
```

		requested function.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.
4	RXMACRO_ALREADY_EXISTS	Duplicate functions cannot be loaded from a macrospace file.
5	RXMACRO_FILE_ERROR	An error occurred accessing a macrospace file.
6	RXMACRO_SIGNATURE_ERROR	A macrospace save file does not contain valid function images.

RexxLoadMacroSpace - Remarks

When *FuncCount* is zero or *FuncNames* is NULL, RexxLoadMacroSpace loads all REXX procedures from the saved file.

If a RexxLoadMacroSpace call would replace an existing macrospace REXX procedure, the entire load request is discarded and the macrospace remains unchanged.

RexxQueryMacro

RexxQueryMacro searches the macrospace for a specified function.

Topics:

- Syntax
- Parameters
- Returns

RexxQueryMacro - Syntax

RexxQueryMacro(FuncName, Position)

RexxQueryMacro - Parameters

FuncName *(PSZ) - input*

Address of an ASCIIZ function name.

Position *(PUSHORT) - output*
Address of an unsigned short integer flag. If the function is loaded in the macrospace, *Position* is set to the current function search-order position.

RexxQueryMacro - Returns

The RexxQueryMacroSpace return values are:

0	RXMACRO_OK	The call to the function completed successfully.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.

RexxReorderMacro

RexxReorderMacro changes the search order position of a loaded macrospace function.

Topics:

RexxReorderMacro - Syntax

RexxReorderMacro(FuncName, Position)

RexxReorderMacro - Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ macrospace function name.

Position *(ULONG) - input*

New search-order position of the macrospace function. Possible values are:

RXMACRO_SEARCH_BEFORE
: The REXX interpreter locates the function before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
: The REXX interpreter locates the function after any registered functions or external REXX files.

RexxReorderMacro - Returns

The RexxReorderMacro return values are:

0	RXMACRO_OK	The call to the function completed successfully.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.
8	RXMACRO_INVALID_POSITION	An invalid search-order position request flag was used.

Macrospace Interface Returns

Return codes for the External Functions Interface are as follows. Each code indicates the cause of a failure in its respective function:

0	RXMACRO_OK	The call to the function completed successfully.
1	RXMACRO_NO_STORAGE	There was not enough memory to complete the requested function.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.
3	RXMACRO_EXTENSION_REQUIRED	An extension is required for the macrospace file name.
4	RXMACRO_ALREADY_EXISTS	Duplicate functions cannot be loaded from a macrospace file.
5	RXMACRO_FILE_ERROR	An error occurred accessing a macrospace file.
6	RXMACRO_SIGNATURE_ERROR	A macrospace save file does not contain valid function images.

```
 7      RXMACRO_SOURCE_NOT_FOUND            The requested file was not
                                            found.

 8      RXMACRO_INVALID_POSITION            An invalid search-order
                                            position request flag was
                                            used.
```

Example

The following is a sample of macrospace usage:

Sample macrospace usage

```
                                    /* first load entire package  */
  RexxLoadMacroSpace(0, NULL, "EDITOR.MAC");
  for (i = 0; i < MACRO_COUNT; i++) {  /* verify each macro          */
                                    /* if not there               */
    if (RexxQueryMacro(macroÝi¨, &position))
                                    /* add to list                */
      RexxAddMacro(macroÝi¨, macro_filesÝi¨,
          RXMACRO_SEARCH_BEFORE);
  }
                                    /* rebuild the macrospace     */
  RexxSaveMacroSpace(0, NULL, "EDITOR.MAC");
.
.
.

                                    /* build the argument string  */
  MAKERXSTRING(argv[0], macro_argument,
      strlen(macro_argument));
                                    /* set up default return      */
  MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));
                                    /* set up for macrospace call */
  MAKERXSTRING(macrospace[0],NULL, 0);
  MAKERXSTRING(macrospace[1],NULL, 0);
  return_code = RexxStart(1,        /* one argument               */
                          argv,     /* argument array             */
                          macro[pos],  /* REXX procedure name     */
                          macrospace,  /* use macrospace version  */
                          "Editor",    /* default address name    */
                          RXCOMMAND,   /* calling as a subcommand */
                          NULL,     /* no exits used              */
                          &rc,      /* converted return code      */
                          &retstr);    /* returned result         */
```

Sample REXX Programs

The following sample programs are included with REXX as .CMD files. Using the online version of this guide, you can view these samples from the OS/2 Toolkit REXX Samples directory. To view a sample REXX program, click on a highlighted program name. The command file will be displayed using the OS/2 system editor.

CCREPLY.CMD
: A concurrent programming example.

  This program demonstrates how to use reply to run two methods at the same time.

COMPLEX.CMD
A complex number class.

This program demonstrates how to create a complex number class using the ::CLASS and ::METHOD directives. An example of subclassing the complex number class (the Vector subclass) is also shown. Finally, the Stringlike class demonstrates the use of a mixin to provide some string behavior to the complex number class.

FACTOR.CMD
A factorial program.

This program demonstrates a way to define a factorial class using the subclass method and the *.methods* environment symbol.

GREPLY.CMD
An example contrasting the GUARDED and UNGUARDED methods.

This program demonstrates the difference between GUARDED and UNGUARDED methods with respect to their use of the object variable pool.

GUESS.CMD
An animal guessing game.

This sample creates a simple node class and uses it to create a logic tree. The logic tree is filled in by playing a simple guessing game.

KTGUARD.CMD
A GUARD instruction example.

This program demonstrates the use of the START method and the GUARD instruction to control the running of several programs. In this sample, the programs are controlled by one 'guarded' variable.

MONTH.CMD
An example that displays the days of the month for January 1994.

This program is similar to the MONTH1.CMD exec found in the *OS/2 2.0 REXX User's Guide.* This version demonstrates the use of arrays to replace stems.

PIPE.CMD
A pipeline implementation.

This program demonstrates the use of ::CLASS and ::METHOD directives to create a simple implementation of a CMS-like pipeline function.

QDATE.CMD
An example that types/pushes today's date and moon phase, in English.

QTIME.CMD
An example that lays or stacks time in real English, and also chimes.

SEMCLS.CMD
An Object REXX semaphore class.

This file implements a semaphore class in Object REXX.

STACK.CMD
A stack class.

This program demonstrates how to implement a stack class using ::CLASS and ::METHOD directives. Also included is a short example of the use of a stack.

USECOMP.CMD
A simple demonstration of the complex number class.

This program demonstrates use of the ::REQUIRES directive, using the complex number class included in the samples.

USEPIPE.CMD
Sample uses of the pipe implementation in PIPE.CMD.

This program demonstrates how you could use the pipes implemented in the pipe sample.

Notices

August 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION, INCLUDING ANY SAMPLE APPLICATION PROGRAMS, "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM authorized reseller or IBM marketing representative.

Copyright Notices

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Note to U.S. Government Users--Documentation related to restricted rights--Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

```
IBM
Object REXX
Operating System/2
OS/2
Presentation Manager
SOMobjects
Workplace Shell
```

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

(No title)

RESULT is also set by an EXIT or REPLY instruction.

(No title)

Float values with maximum precision require NUMERIC DIGITS 15.

(No title)

Programs must provide the hexadecimal value because no conversion is done.
